Automatically Improving Accuracy for Floating Point Expressions

Abstract

Scientific and engineering applications depend on floating point arithmetic to approximate real arithmetic. Unfortunately, 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* improves floating point accuracy by searching for error-reducing transformations. Herbie estimates and localizes rounding error, applies a database of rules to generate improvements, takes series expansions, and combines improvements for different input regions. We evaluated Herbie on every example from a chapter in 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 an average performance overhead of 11%. Colleagues in machine learning have applied Herbie to significantly improve the results of a clustering algorithm, and a mathematical library has accepted a patch generated using Herbie.

1. Introduction

Floating point rounding errors are notoriously difficult to detect and debug [Kahan 2000; Kahan and Darcy 1998; Toronto and McCarthy 2014]. Rounding errors have lead to irreproducibility and even retraction of scientific articles [Altman et al. 2003; Altman and McDonald 2003, 1999], legal regulations in finance [European Commission 1998], and distorted stock market indices [McCullough and Vinod 1999; Quinn 1983]. Many applications which must produce accurate results, including physical simulators and statistical packages, depend on floating point arithmetic to approximate computations over real numbers. Such approximations make these computations feasible, but they also introduce rounding error, which can lead to unacceptable *accuracy*: the approximate results may differ from the ideal real number results by an unacceptable margin.

When these floating point inaccuracies are discovered, many developers first try perturbing the code until the answers produced for problematic inputs are sufficiently accurate [Toronto and McCarthy 2014; Kahan and Darcy 1998]. This process can be tedious and frustrating even when effective, and may only temporarily mask the error if the test inputs are not representative.

Knowledgable developers may turn to formal numerical analysis. The numerical analysis literature includes forward and backward error analysis [Higham 2002; Kahan 1971] to quantify the error of a program, and program transformations [Hamming 1987; Goldberg 1991] which can be applied to improve program accuracy. Unfortunately, these techniques often require understanding the subtle details of floating point arithmetic, and the process is still slow and complex.

Lastly, developers may 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].

As a step toward addressing these challenges, we introduce Herbie, a tool that automatically improves the accuracy of floating point expressions. Herbie searches for error-reducing program transformations by localizing error, generating candidate rewrites, and merging rewrites with complementary effects. Herbie localizes the error of an expression to individual operations by comparing intermediate results for floating point and arbitrary precision evaluations. Herbie then applies a database of rewrite rules and performs series expansion to operations with high error, generating candidate expressions that may provide better accuracy. Finally, Herbie automatically combines improvements for different input regions to produce a single program that improves accuracy across all regions. By combining these techniques, Herbie improves accuracy automatically, without requiring training or manual effort from the programmer.

Herbie complements recent work in analyzing and verifying floating-point programs [Darulova and Kuncak 2014; Barr et al. 2013; Boldo 2009]. 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 improving* the accuracy of input programs. While the transformations Herbie discovers typically correspond to techniques from the numerical methods literature, Herbie does not 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.

[1] Even arbitrary precision floating point can exhibit rounding error if the user selects insufficient precision, and so the developer needs expertise to carefully select a precision that provides sufficient accuracy. (See Section 4.1 and [Toronto and McCarthy 2014].)

We evaluate Herbie on examples drawn from a classic numerical methods textbook [Hamming 1987] 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 11% overhead. 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 have accepted a patch generated using Herbie, which improved the accuracy of a complex number routine.

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

- A method to compute exact floating point prefixes for real number computations.
- A technique for localizing the source of rounding error.
- An algorithm to flexibly apply sequences of rewrites and simplify the results.
- A Laurent series expander which supports transcendental and non-analytic functions.
- An approach to inferring regimes where the error behavior of a program differs.

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 [Anonymized for submission 2014a].

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 quantifies 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.[2] 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.

Floating point operations use a *rounding mode*,[3] 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 also guarantee that $\mathsf{F}(\mathsf{R}(x)) = x$ and that $\mathsf{R}(\mathsf{F}(x))$ is "close" 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} 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. We write $\mathsf{F}(x) = x + x\epsilon_x$, where ϵ_x is the floating point conversion error,[4] and is of absolute value less than 2^{-k}, and where applications of F to different inputs will result in different errors ϵ_x.

Primitive arithmetic operators on floating point numbers such as addition and multiplication are guaranteed to produce accurate 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 value $x + y + (x + y)\epsilon_{x+y}$.

Operators such as exponentiation and trigonometric functions are usually not computed in hardware and must be implemented by libraries.[5] Due to the table maker's dilemma [Lefévre and Muller 2003], these more complex 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 u 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 $e^x + ue^x\epsilon_{e^x}$. Typically, u is less than 8, guaranteeing that all but the two or three least-significant bits are correct.

[2] 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 $\pm m2^{-l/2}$.

[3] 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.

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

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

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 largely accurate, formulas that combine these operators can still be inaccurate. For example, consider the expression $(x+1)-x=1$. The addition introduces error ϵ_2 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 terms ϵ_2 and $(x+1)\epsilon_1\epsilon_2$ are small compared to the true value 1, but $(x+1)\epsilon_1$ 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.

For $-1 < x < 1$, this expression exhibits little error, but as x grows larger, the error grows as well. In general, complex 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 rely on identities of real-number arithmetic. For example, the identity $(x+y)-z=x+(y-z)$ is true of real numbers and allows converting $(1+x)-x$ into $1+(x-x)$ and then into 1, which is exactly accurate. Note that these identities are *false* for floating-point arithmetic, which is why they change the floating-point results and improve accuracy.

The necessary rewrites can be unintuitive. Richard Hamming provides the example [Hamming 1987] of computing $\sqrt{x+1}-\sqrt{x}$. The naive encoding is inaccurate for large positive numbers due to cancellation. Hamming's solution involves rearranging this expression into $1/(\sqrt{x+1}+\sqrt{x})$ using the difference-of-squares formula. Herbie's heuristic search is well-suited to finding such rearrangements.

3. Overview

Consider the familiar quadratic formula:

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

In this form, found in high-school algebra textbooks, the formula is inaccurate for negative b and large positive b. This expression has 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. Herbie can automatically produce a more accurate version using all of Herbie's major subsystems (see Figure 1).

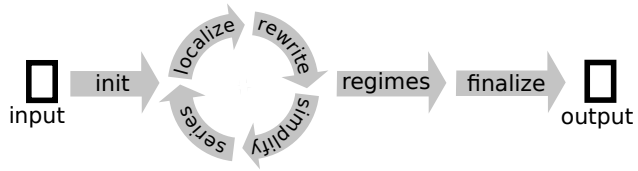

Figure 1. Herbie's process for improving program accuracy.

For negative b, the error is caused by cancellation at the subtraction in the numerator $(-b-\sqrt{b^2-4ac})$. For b^2 much larger than a or c, the descriminant $\sqrt{b^2-4ac}$ 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

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

Other applicable rewrite rules produce ten other rewritten programs. So far, none of these rewrites have 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 find this cancellation (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 subexpression of an expression under 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 rather 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 4.8) 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 4.6) combines the three candidates by inferring an **if** statement to select between them depending on the value of b. The final program produced by Herbie[6] is

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

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 [Goldberg 1991; Hamming 1987].

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[7]. Herbie uses 256 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. However, the real-number semantics is not executable, since exact real numbers cannot be efficiently represented on a computer. Instead, Herbie uses arbitrary precision floating point using GNU MPFR [Fousse et al. 2007].

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 integer arithmetic, the difference is that the precision of integers can be dynamically expanded as necessary, but arbitrary-precision floating-point precision needs to be statically 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 working precision further increases do not change the computed answer for *any* sampled input point, to 64 bits. 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).

[6] Because Herbie is based on sampling and takes into account the lower-level details of floating-point arithmetic, it produces more-complex programs that are somewhat more accurate that this example.

[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(locations(candidate)).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 Herbie, N and M are constants set to 3 and 4 4.3.

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 [Toronto and McCarthy 2014]. We follow STOKE [Schkufza et al. 2014] in defining error as the number of floating-point values between the approximate and exact answers:

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

Intuitively, this counts the number of most-significant bits that the approximate and exact result agree on. 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[8]. Avoiding such rules was usually easy, but required some care to avoid false "identities" such as $\sqrt{x^2} = x$, which is only true for positive x.

Herbie uses a greedy, hill-climbing search to apply this database of rules: it tracks a set of candidate programs, it applies rules at various locations in these candidates, evaluates each resulting program, repeating the process on the best candidates. However, a naive implementation of this method 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. 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 Herbie's main loop finishes, regime inference is used 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.

```
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.

4.3 Localizing error

The search space that Herbie faces is exponential; for large programs, many rewrites are possible, and Herbie must identify which are likely to improve accuracy to prune this search space. 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).

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

```
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 all 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 focuses its rewrites on the 4 operations with the highest local error; our experiments have found no improvement from including more operations, and localizing to more operations slows Herbie down, since more rewrites are considered.

4.4 Recursive rewrite pattern matching

After localizing the error to an operation, Herbie applies rewrites at that operation using its database of rewrite rules. Each rewrite rule might change that operator to a different way of computing the same value, one which is more accurate. One approach would be to simply apply each matching rule; however, this would not 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 to have the most 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 under a common denominator, and then the numerator must be simplified.

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,

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

```
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)
   if is-commutative(top-operator(expr)) :
      return max(sub-iters) + 2
   else :
      return max(sub-iters) + 1
```

Figure 5. Herbie simplifies expressions by creating an equivalence graph [Nelson 1979], and applying rewrite rules at all nodes in the graph. The number of times each rewrite rule is applied depends on the depth 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 only simplifies the children of the node which was most recently rewritten.

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 automatically rewriting each subexpression of an expression, recursively, to match its associated pattern in the rule. 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 making changes at the source of error is ill suited to this task, 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.

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 an equivalence graph [Nelson 1979] 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 to reduce of the total size of the expression, including removing function inverses (as in $\sqrt{x}^2 \rightsquigarrow x$), cancelling like terms (as in $x - x \rightsquigarrow 0$), and combining terms (as in $x + x \rightsquigarrow 2x$).

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 only contain 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 uses a simple bound (see iters-needed in Figure 5) on the number of rewrites necessary to cancel any two terms anywhere in the expression. This makes simplification faster while still allowing all of its important functions.

4.6 Regime inference

Often no candidate programs 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 input regions, which we call *regimes*. Herbie's *regime inference* algorithm automatically detects which programs to use on which inputs. Regime inference prevents Herbie from improving accuracy while producing a program less-accurate on some inputs.

Herbie finds the optimal set of branches using a variant of the Segmented Least Squares dynamic programming algorithm of Kleinberg and Tardős [Kleinberg and Tardős 2005]. 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

```
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 a binary-search.

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 computational cost of adding a branch against the potential benefit of doing so: branches are computationally expensive, but can improve accuracy. To avoid overfitting and generating slow programs, regime inference will only add branches which improve average accuracy across the entire input range by at least one bit.

4.7 Candidate programs table

Between iterations of the core loop, Herbie prunes the set of candidate programs to those which are most accurate on at least one sample point. This keeps exactly those programs which will be useful for regime inference, allowing Herbie to explore programs that improve accuracy only on parts of the input space. 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 result in a greater accuracy of output programs than running for 3 iterations.

Herbie stores the set of candidate programs as a pair of maps: a map from points to a set of alternatives that do best

at that point, and a map from alternatives to the points they do 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 adding a candidate, some existing candidate is no longer best at that point, so it may be possible to prune candidates. Each candidate corresponds to a set of points: the set of points it is best at.

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 turns into an instance of Set Cover. This is the Set Cover problem, which is known to be NP-hard. Herbie uses a variant of the $O(\log n)$ Set Cover approximation algorithm [Chvatal 1979] to solve this problem. There are often points with a unique best candidate at that point; 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—we have never seen a candidate set of more than 28 programs during a normal execution, even though as many as 285 programs are generated.

4.8 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 close to accurate. Such approximations improve accuracy in many cases, and often help avoid over- and under-flow. Regime inference ensures that polynomial approximations are only when accurate.

Herbie's series expansion procedure is a symbolic execution: each variable or constant corresponds to a trivial series, and each function application combines the series expansions of its arguments as dictated by standard mathematical formulas. Both series expansions at 0 and at ∞ are used. 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}; is called a bounded Laurent series, and allows handling expressions such as $\frac{1}{x} - \cot x$ Each coefficient c_n is a symbolic expression, so for expressions with essential singularities at 0, like $e^{1/x}$, the part with a singularity is placed into c_0. Series expansions in multiple variables use a multivariate series. When truncating series, Herbie uses the three nonzero terms with the smallest total degree; we've found this sufficient for the regimes that series expansions are used in.

5. Case Studies

This section describes two examples where Herbie improved the accuracy of real-world programs: in one case, 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 [de Jong 2013]. Among other functions, Math.js supports operations on complex numbers. To compute the square root of a complex number $x + iy$, Math.js used the expression $\frac{1}{2}\sqrt{2(\sqrt{x^2+y^2}+x)}$, which is the standard mathematical definition of complex square roots. However, for large values of x (especially when y is small), this expression is inaccurate. Herbie synthesized a more-accurate form of this expression, which for negative x computes

$$|y|/\sqrt{2\sqrt{x^2+y^2}-x}$$

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 [Anonymized for submission 2014b].

Probabilities in a Clustering Algorithm Herbie has also been useful to practitioners who are not directly interested numerical accuracy. A machine learning colleague 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

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

Our estimates suggest that this simple encoding produces seventeen bits of error, averaged over double-precision floating point values. In an attempt to improve 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.

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

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

Herbie produced superior results with no need for manual algebraic manipulation.

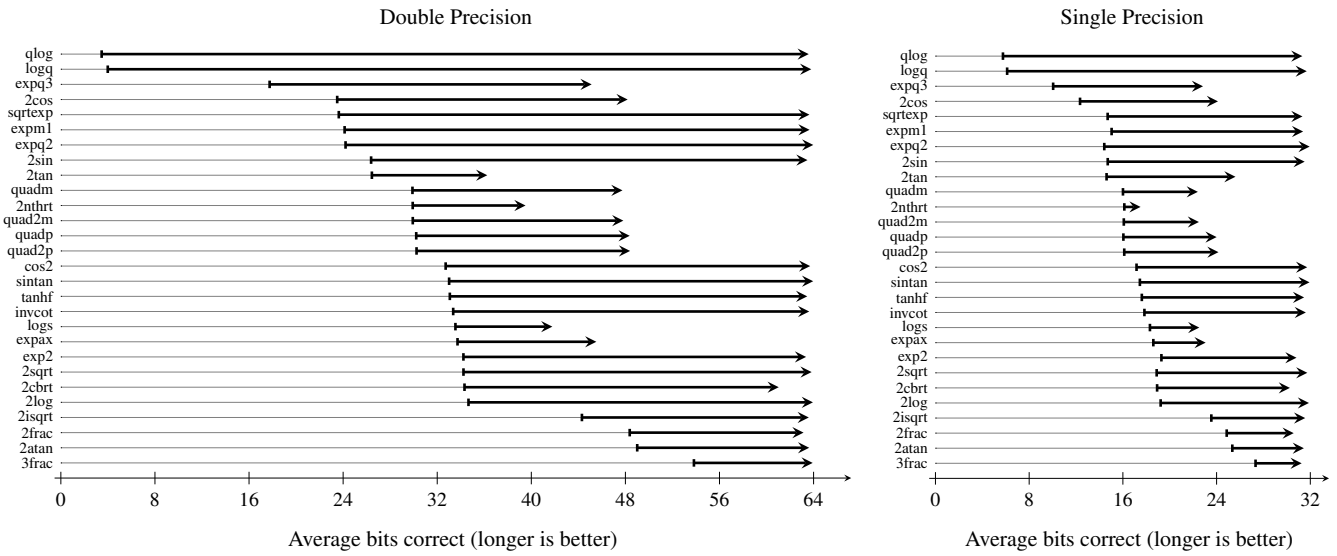

Figure 7. Each row represents the improvement in accuracy achieved by Herbie on a single benchmark. The thick arrow starts at the accuracy of the input program, and ends at the accuracy of Herbie's output. Accuracy is measured by the number of correct output bits, averaged across 100,000 random input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

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* [Hamming 1987], a standard numerical analysis textbook focusing on applying formal numerical analysis to scientific computation. 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. Four of the problems and examples are from the introductory section, 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 transcribed each example directly to Herbie's input format 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. 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.

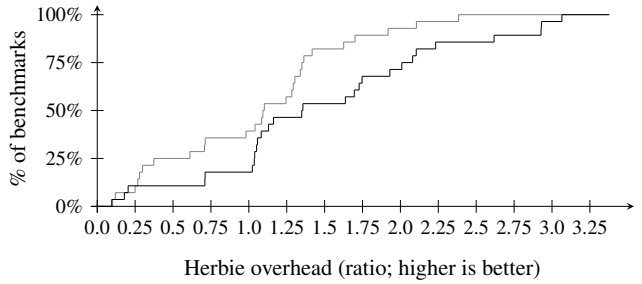

Figure 8. Cumulative distribution of the slow-down from using Herbie. The horizontal axis shows the ratio between the run-time of the input and output programs. The black line is the overhead in Herbie's standard configuration; the gray line is the overhead when regime inference is disabled.

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 this 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 11% slower.

Error Estimation. Each program was run on 100,000 points drawn randomly from the set of double-precision floating point valuesResults were compared with a ground truth computed via the MPFR arbitrary-precision library [Fousse et al. 2007], 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.

Herbie uses average error because it enables our heuristic search to automatically discover classical numerical analysis techniques. However, 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 a handful of test cases by enumerating *all* single-precision floating-point numbers. In some cases, improve 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. Similar results were seen for double-precision, where we used sampling to estimate maximum error.

6.2 Error Evaluation

The figures above compute average error by sampling $100,000$ points and computing the average error across the sample. Since each sampled point has between 0 and 64 bits in error, the standard error of a single sample is at most 64 bits. So estimating accuracy by sampling 100,000 points leads, by the Central Limit Theorem, to an estimate with standard error less than $64/\sqrt{100,000} \approx 0.2$ bits[9]. Thus, improvement by at least a bit (as in every improved test case in Figure 7) represents an improvement by five times the standard error (5σ).

To ensure that sufficiently many bits were used for computing the ground truth exact outputs, we compared the computed ground truth to an evaluation using 65,536-bit precision. In every instance, Herbie's estimate of the number of bits necessary was sufficient to give identical answers to 65,536-bit precision, demonstrating that our ground truth was computed correctly.

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 Herbie in its default configuration. Regime inference helps improve the accuracy of programs on 17 of the 28 benchmarks;

[9] We also repeated the sampling procedure 100 times and computed the standard error of the results, which was an order of magnitude smaller than this conservative upper bound. These results imply that our measurement of accuracy is extremely accurate.

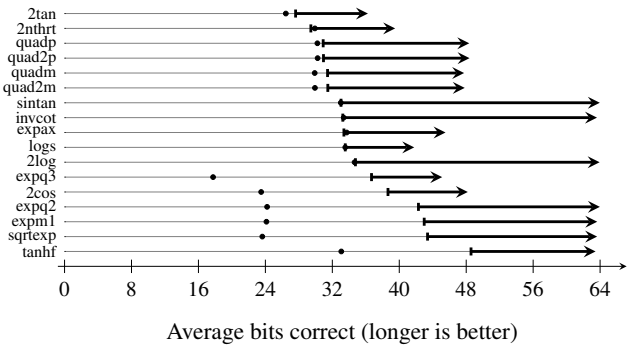

Figure 9. Each arrow represents one of the 17 programs where regime inference improves accuracy; the arrow points from the accuracy without regime inference to the accuracy with regime inference enabled. A dot is drawn at the accuracy of the original program; note that in many of the cases, Herbie is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

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.

However, the branches added by regime inference also make the resulting programs slower (see Figure 8). Regime inference added a median overhead of 7%. Regime inference introduces branches of the form $x_i < c$, where x_i is a program variable and c is a constant. When evaluating performance overhead, inputs are chosen randomly, so these branches are impossible for the processor to predict. This results in a worst case overhead. In practical use, where inputs are not chosen randomly, the overhead of regime inference may be lower.

6.4 Extensibility

Real-world computations are likely to involve functions which Herbie does not understand, or about which Herbie's rule database does not contain necessary rules. So, it is important that Herbie support user extensibility. We made two tests of Herbie's extensibility: we test that the user can add rules to solve examples that Herbie doesn't solve by default; and we test 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.

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 complex and less structured than the problems in NMSE. Herbie has proven success in improving the accuracy of such programs (see Section 5), but we wanted to make 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. For 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 disspell 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.

7. Related Work

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 [Goldberg 1991; Kahan 2000], and in common textbooks [Hamming 1987; Higham 2002]. Herbie uses the techniques invented in this literature, but rearranges formulas automatically, avoiding the need for an expert. It is also difficult to determine the working precision necessary to accurately evaluate a function [Lefévre and Muller 2003]. Recent work on this problem allowed the creation of MPFR, an arbitrary-precision floating point library with correct rounding [Fousse et al. 2007]. Herbie uses MPFR internally to exactly evaluate expression.

Verification Floating point computation is defined in the IEEE 754 standard [IEEE 2008]. However, verification is difficult as programming languages often do not require adherence to this standard [Monniaux 2008]. Programs for computing discriminants [Boldo 2009] and solving simple partial differential equations [Boldo et al. 2013] have recently been verified. Automatic proofs have been also explored: Rosa [Darulova and Kuncak 2014] uses an SMT solver to automatically prove error bounds, while Ariadne [Barr et al. 2013] 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 [Goubault and Putot 2011], and FPDebug uses a dynamic execution with shadow variables in higher precision [Benz et al. 2012].

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. The Stoke super-optimizer supports optimizing floating point programs while guaranteeing that the resulting accuracy is acceptable [Schkufza et al. 2014]. Precimonious [Rubio-González et al. 2013] attempts to decrease the precision of intermediate results to lower run-time and memory use. Neither Stoke nor Precimonious improve floating point accuracy.

Program Transformations M. Martel proposed a bounded brute force search for algebraically-equivalent programs for which a better accuracy bound could be statically proven [Ioualalen and Martel 2013; Martel 2009]. The bounded brute-force search limits the program transformations that can be found. Only addition and multiplication operators are supported. Genetic programming and SMT synthesis have also been explored for synthesizing fixed point programs for evaluating polynomial expressions [Darulova et al. 2013; Eldib and Wang 2013]. Herbie uses a variety of analyses to prune and direct the search for more accurate candidate programs, allowing deeper search and support for a rich set of transcendental functions.

8. Future Work

In the near term, Herbie's techniques can be extended to support for complex numbers and matrices. Herbie could be integrated into a compiler pass, allowing it to transparently improve the accuracy of floating-point programs during compilation. In the longer term, Herbie's techniques could be adapted to reduce error accumulation within loops.

References

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

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.

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

Anonymized for submission. Herbie, 2014a. URL removed.

Anonymized for submission. Numerical imprecision in complex square root, 2014b. URL removed.

E. T. Barr, T. Vo, V. Le, and Z. Su. Automatic detection of floating-point exceptions. POPL '13, 2013. doi: 10.1145/2429069.2429133.

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. doi: 10.1145/2254064.2254118. URL http://doi.acm.org/10.1145/2254064.2254118.

S. Boldo. Kahan's algorithm for a correct discriminant computation at last formally proven. *IEEE Transactions on Computers*, 58(2):220–225, Feb. 2009. ISSN 0018-9340. doi: 10.1109/TC.2008.200. URL http://hal.inria.fr/inria-00171497/.

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. doi: 10.1007/s10817-012-9255-4. URL http://hal.inria.fr/hal-00649240/en/.

V. Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3):pp. 233–235, 1979. ISSN 0364765X. URL http://www.jstor.org/stable/3689577.

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. doi: 10.1145/2535838.2535874. URL http://doi.acm.org/10.1145/2535838.2535874.

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.

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

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

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.

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.

D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, Mar. 1991. ISSN 0360-0300. doi: 10.1145/103162.103163. URL http://doi.acm.org/10.1145/103162.103163.

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.

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

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

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

A. Ioualalen and M. Martel. Synthesizing accurate floating-point formulas. ASAP '13, pages 113–116, June 2013. doi: 10.1109/ASAP.2013.6567563.

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

W. Kahan. Miscalculating area and angles of a needle-like triangle. Mar. 2000. URL http://www.cs.berkeley.edu/~wkahan/Triangle.pdf.

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.

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

V. Lefévre and J.-M. Muller. The table maker's dilemma: our search for worst cases. Oct. 2003. URL http://perso.ens-lyon.fr/jean-michel.muller/Intro-to-TMD.htm.

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

B. D. McCullough and H. D. Vinod. The numerical reliability of econometric software. *Journal of Economic Literature*, 37(2):633–665, 1999. doi: 10.1257/jel.37.2.633.

D. Monniaux. The pitfalls of verifying floating-point computations. *ACM Trans. Program. Lang. Syst.*, 30(3):12:1–12:41, May 2008. ISSN 0164-0925. doi: 10.1145/1353445.1353446. URL http://doi.acm.org/10.1145/1353445.1353446.

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

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

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. doi: 10.1145/2503210.2503296.

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

N. Toronto and J. McCarthy. Practically accurate floating-point math. *Computing in Science Engineering*, 16(4):80–95, July 2014. ISSN 1521-9615. doi: 10.1109/MCSE.2014.90.