Combining Tools for Optimization and Analysis of Floating-Point Computations

Heiko Becker[1], Pavel Panchekha[2], Eva Darulova[1], and Zachary Tatlock[2]

[1] MPI-SWS {hbecker,eva}@mpi-sws.org
[2] University of Washington {pavpan,ztatlock}@cs.washington.edu

Abstract. Recent renewed interest in optimizing and analyzing floating-point programs has lead to a diverse array of new tools for numerical programs. These tools are often complementary, each focusing on a distinct aspect of numerical programming. Building reliable floating point applications typically requires addressing several of these aspects, which makes easy composition essential. This paper describes the composition of two recent floating-point tools: Herbie, which performs accuracy optimization, and Daisy, which performs accuracy verification. We find that the combination provides numerous benefits to users, such as being able to use Daisy to check whether Herbie's unsound optimizations improved the worst-case roundoff error, as well as benefits to tool authors, including uncovering a number of bugs in both tools. The combination also allowed us to compare the different program rewriting techniques implemented by these tools for the first time. The paper lays out a road map for combining other floating-point tools and for surmounting common challenges.

1 Introduction

Across many domains, numerical computations specified over the reals are actually implemented using floating-point arithmetic. Due to their finite nature, operations on floating-point numbers cannot be calculated exactly and accumulate roundoff errors. In addition, real-valued identities such as associativity no longer hold, making manual reasoning and optimization challenging. To address these challenges, new automated tools have recently been developed which build on advances in program rewriting and verification techniques to enable even non-experts to analyze and optimize their floating point code.

Some of these tools use sound techniques to statically bound roundoff errors of straight-line floating-point programs [9,12,22,15,16,8] and partially automate complex analysis tasks [18,10]. Other such tools use dynamic techniques to find inputs that suffer from large rounding errors [4,23]. Yet other tools perform rewriting-based optimization [5,17,20,8] and mixed-precision tuning [3,7] to improve the accuracy and performance of floating-point programs.

Since these tools are typically complementary, each focusing on a distinct aspect of numerical reliability, users will need to compose several to meet their development needs. This makes ease of composition essential, and some first

steps in this regard have been taken by the FPBench project [6], which provides a common specification language for inputs to floating-point analysis tools similar to the one provided by the SMT-LIB standard [1]. However, no literature yet exists on the actual use of FPBench to compose tools and on the challenges that stand in the way of combining different floating-point tools, such as differing notions of error and different sets of supported functions.

In this paper we report on our experience implementing the first combination of two complementary floating-point analysis tools using FPBench: Herbie [17] and Daisy [8]. Herbie optimizes the accuracy of straight-line floating-point expressions, but employs a dynamic roundoff error analysis and thus cannot provide sound guarantees on the results. In contrast, Daisy performs static analysis of straight-line expressions, which is sound w.r.t. IEEE754 floating-point semantics [13]. Our combination of the tools is implemented as a script in the FPBench repository (`https://github.com/FPBench/FPBench`).

We see this combination of a heuristic and a sound technique as particularly interesting; Daisy can act as a backend for validating Herbie's optimizations. Daisy computes improved worst-case roundoff error bounds for many (but not all) expressions optimized by Herbie. On others it raises an alarm, discovering division-by-zero errors introduced by Herbie. We also improved the precision of Daisy's analysis of special functions as we found that some were sound but not accurate enough. Thus, the combination was also useful in uncovering limitations of both tools.

Daisy additionally implements a sound genetic programming-based accuracy optimization procedure. Our combination of Daisy and Herbie allows us to compare it to Herbie's unsound procedure based on greedy search. We discover important differences between the two procedures, suggesting that the techniques are not competitive but in fact complementary and best used in combination.

Some of the challenges we encountered, such as differing supported functions and different error measures, are likely to be encountered by other researchers or even end users combining floating-point tools, and our experience shows how these challenges can be surmounted. Our evaluation on benchmarks from the FPBench suite also shows that tool composition can provide end-to-end results not achievable by either tool in isolation and suggests that further connections with other tools should be investigated.

2 Implementation

The high-level goal of our combination is to use Daisy as a verification backend to Herbie to obtain a sound upper bound on the roundoff error of the expression returned by Herbie. By also evaluating the roundoff error of Herbie's output and of the input expression, we can obtain additional validation of the improvement. It should be noted, however, that Daisy cannot verify whether the actual worst-case or average roundoff error has decreased—a decrease in the computed upper bound can be due to an actual decrease or simply due to a stronger static bound.

```
f_res = Herbie(f_src)
err_src = min{ Daisy(A, FPCore2Scala(f_src)) | A <- AnalysisTypes }
err_res = min{ Daisy(A, FPCore2Scala(f_res)) | A <- AnalysisTypes }
```

Fig. 1. Pseudocode of the script used to compose Herbie and Daisy into a single tool; `AnalysisTypes` contains different modes Daisy can be run in.

In many cases, however, such as in safety-critical systems, just *proving* a smaller static bound is already useful.

We have implemented the combination in a script, which we sketch in Figure 1. For each straight-line input program f_{src}, we first run Herbie to compute an optimized version f_{res}. Both the optimized and unoptimized version are translated into Daisy's input format (using `FPCore2Scala`), and Daisy is run on both versions to compute error bounds.

Daisy supports several different types of error analysis, and we run Daisy in a portfolio style, where the tightest bound computed by any of the analyses is used. In particular, we use the interval analysis with subdivisions mode and the SMT solver mode (with Z3 [11] as the solver).[3] Since each analysis is sound, this provides the tightest error bound that Daisy can prove.

When implementing the script that runs Herbie and Daisy together we had to address two major differences between the two tools: Herbie and Daisy use different input (and output) formats, and Daisy requires domain bounds on all input variables, whereas Herbie allows unbounded inputs. While the implemented script is simple, it took several iterations to implement. The most time consuming part was the improvements that only became apparent after running the tools together. We will first explain how we solved the two differences between Daisy and Herbie and then give an overview on the improvements in both tools.

Formats To avoid having to add new frontends, we implemented a translator from FPBench's FPCore format to Daisy's Scala-based input language. As Herbie produces optimized expressions in FPCore, this translator allows us to run Daisy on both the benchmarks and on Herbie's optimized expressions. This translator is now part of the FPBench toolchain and can be used by other researchers and by users to integrate Daisy with other tools developed as part of the FPBench project.

Preconditions Both Daisy and Herbie allow preconditions for restricting the valid inputs to a floating-point computation. For Herbie, these preconditions are optional. In contrast, Daisy requires input ranges for performing a forward dataflow analysis to compute sound absolute roundoff error bounds. Several of the benchmarks in FPBench did not have a specified precondition. For our experiments,

[3] We found that neither interval analysis without subdivision nor alternate SMT solvers provided tighter bounds.

we manually added a meaningful precondition to these programs, with preconditions chosen to focus on input values with significant rounding errors. To avoid biasing the results, the preconditions were simple order-of magnitude ranges for each variable, with the endpoints of these ranges chosen from 1, 10^{10}, or 10^{20} and their inverses and negations.

Improvements in Daisy and Herbie Connecting Daisy and Herbie and running each on several previously unseen benchmarks uncovered numerous possibilities for improvements in Herbie and Daisy.

In Herbie, several bugs were discovered by our efforts: an incorrect type-checking rule for `let` statements (which would reject some valid programs); incorrect handling of duplicate fields (which allowed one field to improperly override another); and an infinite loop in the sampling code (triggered by some preconditions). Real users running older versions of Herbie have since also reported these bugs, suggesting that issues addressed during tool composition helped improve user experience generally.

In Daisy, we discovered that the analysis of elementary functions was unnecessarily conservative and improved the rational approximations used. Error handling in both tools was also improved such that issues like (potential) divisions by zero or timeouts are now accurately reported. This more precise feedback significantly improves user friendliness and reduces debugging time.

3 Experimental Results

We perform two evaluations of our combination of Daisy and Herbie: we first use Daisy as a verification backend for Herbie and then we compare Daisy's and Herbie's rewriting algorithms. Both experiments use all supported benchmarks from the FPBench suite. We give the full table with all the evaluation data in our technical report [2]. The experiments were run on a machine with an i7-4790K CPU and 32GB of memory. For each benchmark we give both Daisy and Herbie a timeout of 10 minutes.

Composing Daisy and Herbie Our first experiment considers Daisy solely as a tool for computing floating-point error bounds. Herbie is then used to attempt to improve the benchmark's accuracy.

Of the 103 benchmarks, Herbie times out on 31 of them. Of the remaining 72 benchmarks, Daisy raises an alarm[4] on 24 and can prove a bound on 48. Of the 48 benchmarks where Daisy can prove an error bound, Daisy's roundoff error analysis can prove a tighter worst-case error bound for 22 of Herbie's outputs, an equal bound for 18, and a looser bound for 8. These results are summarized in the left-most graph in Figure 2.

Of the 24 benchmarks where Daisy raises an alarm, for 13 Daisy raises an alarm on the original input program and for 16 the alarm is raised on the output

[4] Indicating that it could not prove the absence of invalid operations, such as divisions by zero.

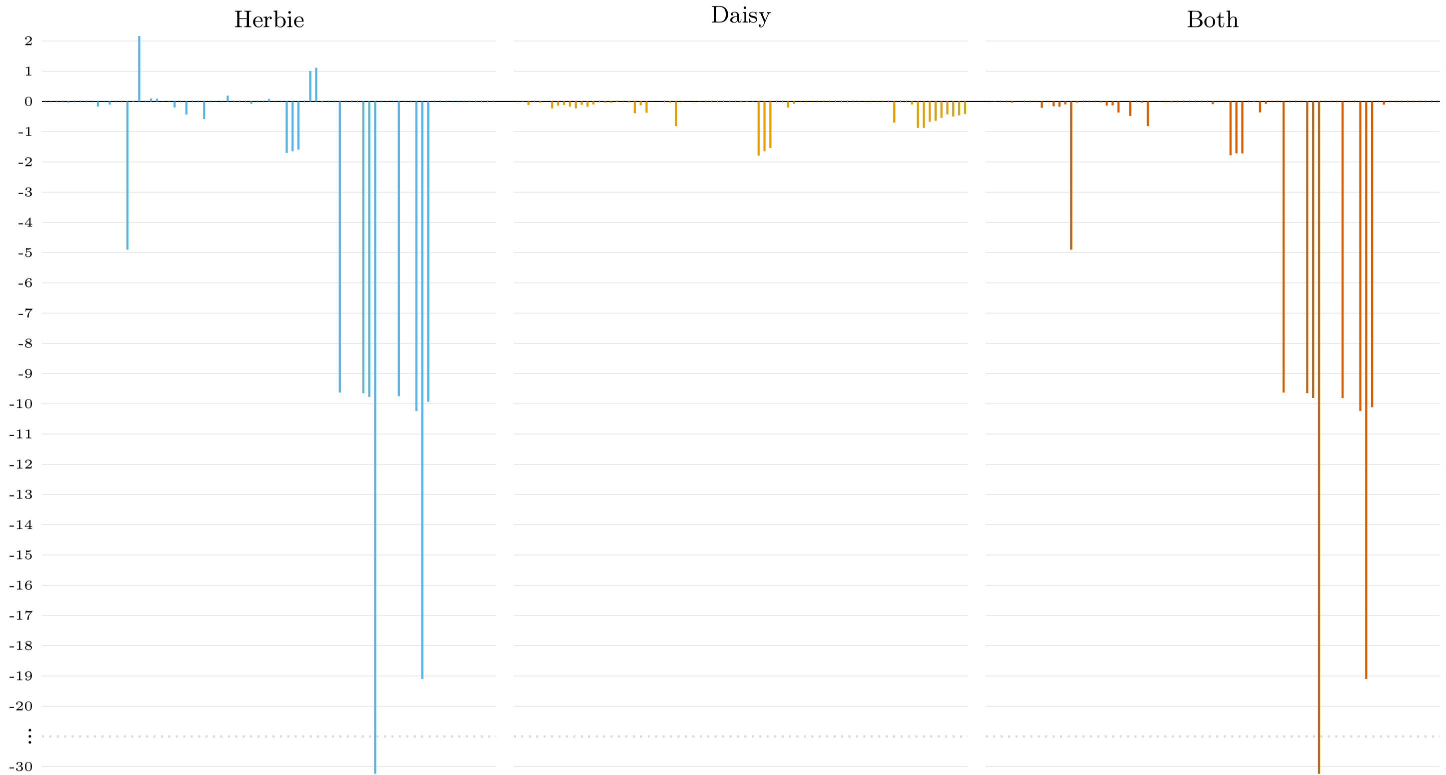

Fig. 2. The orders of magnitude change in Daisy's worst case error estimate after rewriting with Herbie, Daisy, and both Herbie and Daisy (left to right). Note that combining both rewriting algorithms keeps the large beneficial changes introduced by Herbie but avoids its detrimental changes.

program Herbie produced. Some of these alarms are true positives while others are spurious. For example, in some benchmarks Herbie had introduced a possible division by 0, which Daisy was able to detect. In others, the output contained expressions like $1/(x(1+x))$, with $10^{10} < x < 10^{20}$, where Daisy is unable to prove that no division by zero occurs.

We see this as good evidence that Daisy can be used as a verification backend for Herbie. One challenge is that Daisy's error analysis can only show that Herbie's output has a smaller error *bound*, not that it is more accurate at any particular point on the original program. Additionally, it is difficult to determine which of Daisy's alarms are spurious. Despite these challenges, the combination of Daisy and Herbie was able to produce large, verified improvements in rounding error in many benchmarks.

Comparing Daisy's and Herbie's Error Measures One topic of particular interest in combining Daisy and Herbie is their different measures of floating-point error. Herbie measures the error in terms *units in the last place*, or ULPs. To compute this error, Herbie randomly samples input values and takes the average of error across those inputs. It thus provides a *dynamic, unsound measure of average ULPs* of error. Daisy, meanwhile, uses a mathematical abstraction based on the definition of rounding and IEEE754 operator semantics to provide *static, sound bounds on the maximum absolute error.* The relationship between these two measures of error is central to using Herbie and Daisy together.

Despite these stark differences between Herbie's and Daisy's error measures, our evaluation data shows that Daisy and Herbie can be fruitfully used together: Daisy verifies that Herbie's improved program is no less accurate for 40/48 of the benchmarks. This suggests that, though Daisy and Herbie use very different means to measure error, both are successfully measuring the same underlying notion of error. The fact that Daisy's and Herbie's error measures are suited to their particular approaches (static analysis and program search) suggests that future tools should focus not on measuring error "correctly" but on finding an error measure well suited to their technical approach.

Comparing Daisy and Herbie Our second experiment compares Daisy's and Herbie's rewriting algorithms. Daisy uses genetic programming to search for a program with a tighter bound. Herbie, by contrast, uses a greedy search over a suite of different rewriting steps. We compare Herbie's rewriting algorithm, Daisy's rewriting algorithm, and Daisy's rewriting algorithm applied to the results of Herbie's rewriting algorithm. Figure 2 summarizes the accuracy improvements.

Of the 103 benchmarks, at least one of the rewriting algorithms succeeds on 71. Of the 71, Daisy's rewriting algorithm tightens the worst-case bound for 42 benchmarks; Herbie's for 22 benchmarks; and the combination for 34 benchmarks. Furthermore, Herbie's rewriting algorithm loosens the worst-case bound for 8 benchmarks, a consequence of its unsound error measurement technique or differing notion of error, while the combination does so for only 2.

Not only the number but also size of the error improvement matters. Daisy's rewriting algorithm was able to reduce the error bound by a factor of 1.39 (0.14

orders of magnitude) on average; Herbie's by a factor of 13.07 (1.12 orders of magnitude); and the combination by a factor of 15.3 (1.18 orders of magnitude). The combination clearly provided the greatest reduction in error bounds; furthermore, Daisy's algorithm provides larger benefits when applied to Herbie's optimized program than when applied to the benchmark directly.

It seems that Daisy's rewriting algorithm provides a fairly consistent but small tightening of error bounds, while Herbie's algorithm can suggest dramatic and unexpected changes in the expression. However, these large changes sometimes have significantly *looser* error bounds. In those cases, combining Herbie's and Daisy's rewriting algorithms provides a tighter error bound, reaping the benefits of Herbie's rewriting algorithm without the large increases in error bounds that it sometimes causes.

4 Discussion

This paper reports on the combination of Daisy and Herbie and illustrates the benefits of composing complementary floating-point tools to achieve results neither tool provides in isolation. This case study serves as a representative example: similar combinations could be constructed for other tools using this paper's approach. Combinations of Gappa [10], Fluctuat [12], FPTaylor [22], or other verification tools [9,15,16] with Herbie could also allow validating Herbie's optimizations. Verification tools could also be used to validate the output of other unsound tools, such as Precimonious [19] and STOKE [21]. Comparisons with sound optimization tools such as Salsa [5] and FPTuner [3] could also be explored.

Ultimately, we envision using the combination of Daisy and Herbie within larger developments such as VCFloat [18]. VCFloat provides partial automation for reasoning about floating-point computations in CompCert C-light [14] programs. In this context, our toolchain could provide an optimization tactic, that could be applied to (provably) increase accuracy for floating-point segments of C-light programs.

References

1. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.0. In: Gupta, A., Kroening, D. (eds.) International Workshop on Satisfiability Modulo Theories (Edinburgh, UK) (2010)
2. Becker, H., Panchekha, P., Darulova, E., Tatlock, Z.: Combining tools for optimization and analysis of floating-point computations. arXiv preprint arXiv:1805.02436 (2018)
3. Chiang, W.F., Baranowski, M., Briggs, I., Solovyev, A., Gopalakrishnan, G., Rakamarić, Z.: Rigorous Floating-Point Mixed-Precision Tuning. In: Symposium on Principles of Programming Languages (POPL). pp. 300–315. ACM (2017)
4. Chiang, W.F., Gopalakrishnan, G., Rakamaric, Z., Solovyev, A.: Efficient Search for Inputs Causing High Floating-point Errors. In: Symposium on Principles and Practice of Parallel Programming (PPoPP). vol. 49, pp. 43–52. ACM (2014)

5. Damouche, N., Martel, M., Chapoutot, A.: Intra-procedural optimization of the numerical accuracy of programs. In: Formal Methods for Industrial Critical Systems. pp. 31–46. Springer (2015)
6. Damouche, N., Martel, M., Panchekha, P., Qiu, C., Sanchez-Stern, A., Tatlock, Z.: Toward a Standard Benchmark Format and Suite for Floating-Point Analysis. In: International Workshop on Numerical Software Verification. pp. 63–77. Springer (2016)
7. Darulova, E., Horn, E., Sharma, S.: Sound Mixed-Precision Optimization with Rewriting. In: International Conference on Cyber-Physical Systems (ICCPS) (2018)
8. Darulova, E., Izycheva, A., Nasir, F., Ritter, F., Becker, H., Bastian, R.: Daisy - Framework for Analysis and Optimization of Numerical Programs. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (2018)
9. Darulova, E., Kuncak, V.: Towards a Compiler for Reals. ACM Transactions on Programming Languages and Systems (TOPLAS) 39(2), 8 (2017)
10. De Dinechin, F., Lauter, C.Q., Melquiond, G.: Assisted Verification of Elementary Functions using Gappa. In: ACM Symposium on Applied Computing. pp. 1318–1322. ACM (2006)
11. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. TACAS'08/ETAPS'08 (2008), http://dl.acm.org/citation.cfm?id=1792734.1792766
12. Goubault, E., Putot, S.: Robustness Analysis of Finite Precision Implementations. In: Asian Symposium on Programming Languages and Systems (APLAS). pp. 50–57. Springer (2013)
13. IEEE, C.S.: IEEE Standard for Floating-Point Arithmetic. IEEE Std 754-2008 (2008)
14. Leroy, X.: Formal Verification of a Realistic Compiler. Communications of the ACM 52(7) (2009)
15. Magron, V., Constantinides, G., Donaldson, A.: Certified Roundoff Error Bounds Using Semidefinite Programming. ACM Transactions on Mathematical Software 43(4), 1–34 (2017)
16. Moscato, M., Titolo, L., Dutle, A., Munoz, C.A.: Automatic Estimation of Verified Floating-Point Round-Off Errors via Static Analysis. In: International Conference on Computer Safety, Reliability, and Security. pp. 213–229. Springer (2017)
17. Panchekha, P., Sanchez-Stern, A., Wilcox, J.R., Tatlock, Z.: Automatically Improving Accuracy for Floating Point Expressions. In: Conference on Programming Language Design and Implementation (PLDI) (2015)
18. Ramananandro, T., Mountcastle, P., Meister, B., Lethin, R.: A Unified Coq Framework for Verifying C Programs with Floating-Point Computations. In: Certified Programs and Proofs (CPP). pp. 15–26. ACM (2016)
19. Rubio-González, C., Nguyen, C., Nguyen, H.D., Demmel, J., Kahan, W., Sen, K., Bailey, D.H., Iancu, C., Hough, D.: Precimonious: Tuning assistant for floating-point precision. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. pp. 27:1–27:12. SC '13, ACM (2013)
20. Sanchez-Stern, A., Panchekha, P., Lerner, S., Tatlock, Z.: Finding Root Causes of Floating Point Error with Herbgrind. arXiv preprint arXiv:1705.10416 (2017)
21. Schkufza, E., Sharma, R., Aiken, A.: Stochastic optimization of floating-point programs with tunable precision. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 53–64. PLDI '14, ACM (2014)

22. Solovyev, A., Jacobsen, C., Rakamarić, Z., Gopalakrishnan, G.: Rigorous Estimation of Floating-Point Round-off Errors with Symbolic Taylor Expansions. In: International Symposium on Formal Methods (FM). pp. 532–550. Springer (2015)
23. Zou, D., Wang, R., Xiong, Y., Zhang, L., Su, Z., Mei, H.: A Genetic Algorithm for Detecting Significant Floating-point Inaccuracies. In: IEEE International Conference on Software Engineering (ICSE). vol. 1, pp. 529–539. IEEE (2015)