Scaling up Roundoff Analysis of Functional Data Structure Programs

Anastasia Isychev[1] [0000−0001−6375−0421] ✉ and
Eva Darulova[2] [0000−0002−6848−3163]

[1] TU Munich, Germany izycheva@in.tum.de
[2] Uppsala University, Sweden eva.darulova@it.uu.se

Abstract. Floating-point arithmetic is counter-intuitive due to inherent rounding errors that potentially occur at every arithmetic operation. A selection of automated tools now exists to ensure correctness of floating-point programs by computing guaranteed bounds on rounding errors at the end of a computation, but these tools effectively consider only straight-line programs over scalar variables. Much of numerical codes, however, use data structures such as lists, arrays or matrices and loops over these. To analyze such programs today, all data structure operations need to be unrolled, manually or by the analyzer, reducing the analysis to straight-line code, ultimately limiting the analyzers' scalability.
We present the first rounding error analysis for numerical programs written over vectors and matrices that leverages the data structure information to speed up the analysis. We facilitate this with our *functional* domain-specific input language that we design based on a new set of numerical benchmarks that we collect from a variety of domains. Our DSL explicitly carries semantic information that is useful for avoiding duplicate and thus unnecessary analysis steps, as well as enabling abstractions for further speed-ups. Compared to unrolling-based approaches in state-of-the-art tools, our analysis retains adequate accuracy and is able to analyze more benchmarks or is significantly faster, and particularly scales better for larger programs.

Keywords: floating-point arithmetic, rounding error, data structures

1 Introduction

Floating-point arithmetic is notorious for being unintuitive due to its special values as well as rounding operations, the latter inevitably introducing errors

at most arithmetic operations. While special values (Not-a-Number and Infinity) can be explicitly and relatively easily detected during a computation, rounding errors are more tricky as, in general, one does not have results of an exact, infinite-precision computation available for comparison. Static analysis techniques that *soundly* bound rounding errors for all possible inputs, i.e. that compute a guaranteed upper bound on the errors, are thus essential especially for safety-critical systems.

Providing a reasonably accurate sound rounding analysis that tightly bounds rounding errors without extensive over-approximation is intricate already for non-linear *straight-line* code, as the number of recent efforts and tools demonstrates [8,30,9,13,31,7]. Effectively analyzing finite precision beyond straight-line code, and scaling to larger programs is an open challenge [8,9].

Much of numerical code operates over data structures such as arrays and matrices using some form of loop, for instance when computing statistics during data analyses, performing signal processing or Fourier and stencil transforms in embedded systems, calculating dot products in neural networks, etc. In principle, it is possible to 'unroll' such operations into straight-line code, by assigning individual array elements to scalar variables and unrolling all loops. Existing tools either expect this transformation to be done manually by the user in a tedious and error-prone process [9], or allow the user to specify programs imperatively as loops over arrays and unroll automatically when instructed to do so [13]. One way or another, the rounding error analysis itself is reduced to one over a potentially huge straight-line program, limiting the analyses' scalability.

The alternative standard approach for verifying programs with arrays that abstracts all data structure elements by a single representative are, in general, unsuitable for finite-precision rounding error analysis. The reason for this is that the magnitude of rounding errors directly and significantly depends on the magnitudes of the program inputs. Abstracting data structure elements of different sizes by a single value, resp. interval, leads to over-approximations of the rounding errors, and thus to inaccurate and possibly unusable error bounds.

This paper presents the first rounding error analysis with *explicit* support for operations and bounded loops over array-like data structures (i.e. vectors or lists and matrices). To facilitate this analysis we design a *functional* domain-specific input language (DSL) with operations over lists and matrices that allows to express many commonly used patterns in numerical computing and that serves as the input to our tool.

The benefit of a functional input language is two-fold. First, it allows users to succinctly express their computations and reduces the possibility of common (off-by-one) indexing errors. More importantly, however, a functional language carries *semantic information* that can be *leveraged* by the analysis, removing the need to unroll many operations. For example, loops applying a function to each value in a list (functional $\texttt{map}(\lambda x.f(x))$) do not propagate errors between iterations, and a rounding error analysis only has to analyze the loop body once. An unrolling of the loop would lose that high-level information and effectively re-compute the analysis for each loop iteration. For operations that do require

unrolling, we show how to use the semantic information to avoid recomputing analysis information that can be effectively over-approximated, further reducing the burden on the analysis. Our abstraction is designed for rounding errors and accounts for different variables' ranges and thus provides a viable tradeoff between analysis accuracy and performance.

We design our input DSL based on a new set of numerical benchmarks that we collected from a variety of domains. We implement our rounding error analysis for this DSL in a tool called DS2L and show that compared to a baseline analysis that unrolls all operations, it can substantially reduce analysis time with little impact on analysis accuracy.

Note that all loops in our DSL are bounded: they are loops over the elements in a data structure (e.g. via higher-order functions). We specifically do not attempt to solve the general problem of rounding errors in unbounded loops [8], but focus on providing an efficient analysis for commonly used operations over data structures.

Our focus is on *scalability* and we thus compare DS2L against the two most scalable (available) rounding error analysis tools Fluctuat [13] and Satire [9]. Our evaluation shows that for benchmarks with large data structure sizes, DS2L scales significantly better: it can analyze many more benchmarks (has fewer timeouts, overflows and infinite error bounds) and is several times faster.

While we evaluate DS2L only on floating-point code to permit a comparison with existing tools, our analysis extends to fixed-point arithmetic as well and DS2L only requires a simple implementation change.

Contributions. In summary, this paper makes the following contributions:

- a new open-source finite-precision benchmark set;
- a fully automated, sound rounding error analysis for programs written in a functional DSL (Section 4);
- an open-source implementation of this analysis (Section 5);
- an evaluation against state-of-the-art analysis tools (Section 6).

The artifact with the benchmark set, DS2L's source code and scripts to run the experiments is available under `https://zenodo.org/record/8179028`, the source code is also available at `https://github.com/malyzajko/daisy`.

2 State-of-the-Art in Rounding Error Analysis

Before we explain our own approach, we first provide background on existing rounding error analysis tools that work for straight-line code with arithmetic operations on *scalar* values. Our own analysis (explained in Section 4) re-uses this baseline for straight-line arithmetic expressions, and we use it also for comparison in the evaluation (Section 6). The vast majority of existing sound rounding error analyses abstract the IEEE-754 [17] floating-point operations with the following equation:

$$x \circ_F y = (x \circ y)(1 + e) + d, |e| \leq \epsilon_M, |d| \leq \delta_M \tag{1}$$

where $\circ \in \{+, -, *, /\}$, $\circ_F$ is the respective floating-point operation, ϵ_M is the machine epsilon and δ_M captures the error due to subnormal numbers. $\epsilon_M = 2^{-53}$ and $\delta_M = 2^{-1075}$ for double-precision floating-point arithmetic that we assume in this paper. $e(x \circ y) + d$ then bounds the worst-case absolute rounding error of an operation $x \circ y$.[3] Errors on individual operations thus depend on the magnitude of the intermediate expressions (such as $x \circ y$), and furthermore propagate through subsequent operations where they may get magnified or diminished, depending on the operation and the ranges.

State-of-the-art rounding error analyses use one of two approaches: dataflow based as implemented in the tools Fluctuat [13] and Daisy [7], or global optimization-based as implemented in FPTaylor [30], Precisa [31] or Satire [9].

The baseline analysis that we choose for straight-line arithmetic operations is of the dataflow type. To compute the overall error, a forward dataflow analysis tracks two abstract domains: one for the (real-valued) ranges at each intermediate operation, and one for the accumulated errors. These are typically computed using interval arithmetic [27] and affine arithmetic [11], respectively. Affine arithmetic can track linear correlations between variables and often (but not always) computes more accurate error bounds than interval arithmetic.

The alternative analysis phrases the computation of the rounding error as a global nonlinear real-valued optimization problem [9,30,31]. We specifically choose a dataflow approach as our base analysis for several reasons. First, it is unclear how to *effectively* use semantic information from the iterators in the global error constraint. Additionally, we identified optimization opportunities when the range information is available separately from the errors. Finally, even though in this paper we focus on floating-point arithmetic for simplicity, dataflow analysis is immediately applicable to fixed-point arithmetic as well, making our analysis more widely applicable. A global symbolic error constraint optimization works well for floats whose dynamic range allows them to represent many values. However, an *efficient usage* of fixed-points requires the integer and fractional bits to be assigned individually for *each subexpression*. To do that, one needs to know the full range of values taken by a (sub-)expression. While it is technically possible to obtain this information also for the symbolic error constraint (for instance, with some other analysis), this incurs significant overhead. Therefore, the global optimization-based approach is only applied to floating-points.

3 DSL for List-like Data Structures

Before designing our functional domain-specific language for numerical computations (Section 3.2), we collected a new set of benchmarks that informed the design of our DSL, and specifically the set of supported operations (Section 3.3).

[3] We compute *absolute* errors. While relative errors may seem a more appropriate error measure and some analyses exist [18,29], in practice their computation is limited to applications where 0 does not occur as a possible value (otherwise leading to undefined errors), severely limiting their applicability.

3.1 Benchmark Set

Rounding error analysis on programs that contain operations on data structures such as arrays and loops over them is an open challenge, and correspondingly there is no standard benchmark set yet. The existing FPBench benchmarks [6] cover only straight-line code and a few while-loops but no data structures. We therefore create a new benchmark set that covers different domains where numerical computations are frequent:

- statistical computations: *avg, stdDeviation, variance*
- linear and nonlinear digital filters: *roux1, goubault, harmonic* and *nonlin{1-3}* [25]
- differential equations: *lorenz, pendulum* [9,8]
- signal processing: *alphaBlending* (image mask), *fftvector, fftmatrix* (two versions of forward Fourier transform)
- stencil computations: *convolve2d_size3, sobel3, heat1d* [9,8]
- neural networks: *lyapunov, controllerTora* [20]

Some of the benchmarks from FPBench contain loop bodies of control loops, which we rephrase as loops over arrays of sensor data. Other benchmarks have been collected from scientific publications [25,9,8,20] as well as open-source implementations in different programming languages.

3.2 A Functional DSL

Many verification techniques face the dilemma of either adapting the techniques to work on legacy code and (possibly) giving up some precision, or requiring to rewrite the code with verification in mind and being able to reason about a program in more detail. In this work, we choose the second option, and note that our domain-specific language uses Scala syntax and is similar to other existing functional languages and we thus expect it to be largely familiar to developers.

The goal of our DSL is to allow a convenient way to 1) write programs that perform operations on array-like data structures and 2) to analyze them. Our main insight is that a functional style of programming covers both aspects: it allows for a more succinct representation of programs and it retains high-level semantic information of the operations that can be leveraged by the analysis.

heat1d Example We illustrate the succinctness of our DSL on one of the benchmarks that we collected from related work [9]. Figure 1 shows the function *heat1d* in the input formats of two different tools. The *heat1d* function takes as input a temperature distribution and computes the temperature at a coordinate `x0` after 32 units of time. The computation requires temperature values for neighboring coordinates which must be repeatedly recomputed, which is essentially a stencil.

The original straight-line version of the *heat1d* benchmark comes from Satire analyzer [9] and includes *1094 lines of code*, 67 of which specify input ranges of (individual) variables, the rest are unrolled loops. Unrolled computations are not only lengthy, but also error-prone and unnatural for a user to write. A more natural choice when implementing the same algorithm in an imperative style

```c
#include <fluctuat_math.h>
#define N 33
// stencil computations
double heat1d(double (*xm)[N], double (*xp)[N], double* x0) {
int i,j;
for(j=1;j<N; j++) {
  for(i=2; i<(N-j); i++) {
    xm[j][i]=0.25*xm[j-1][i+1]+0.5*xm[j-1][i]+0.25*xm[j-1][i-1];
    xp[j][i]=0.25*xp[j-1][i-1]+0.5*xp[j-1][i]+0.25*xp[j-1][i+1];
  }
  xm[j][0] = 0.25*xm[j-1][1] + 0.5*xm[j-1][0] + 0.25*x0[j-1];
  xp[j][0] = 0.25*xp[j-1][1] + 0.5*xp[j-1][0] + 0.25*x0[j-1];
  x0[j] = 0.25*xm[0][j-1] + 0.5*x0[j-1] + 0.25*xp[0][j-1];
}
return x0[N-1];
}
int main() {
  int i,j;
  double x0[N];
  double xm[N][N];
  double xp[N][N];
  // specify input ranges
  for(i=0; i<N; i++){
    x0[i] = DBETWEEN(1.0, 2.0);
    for(j=0; j<N; j++){
      xm[i][j] = DBETWEEN(1.0, 2.0);
      xp[i][j] = DBETWEEN(1.0, 2.0);
  }}
  heat1d(xm, xp, x0);
  return 0;
}
```

(a) Imperative loop (Fluctuat's input)

```
def heat1d(ax: Vector): Real = {
 require(1.0<=ax && ax<=2.0 && ax.size(33))
 if (ax.length() <= 1) {
   ax.head
 } else {
   val coef = Vector(List(0.25, 0.5, 0.25))
   val updCoefs: Vector =
    ax.slideReduce(3,1)(v => (coef*v).sum())
   heat1d(updCoefs)
 }
}
```

(b) Functional style (our DSL)

Fig. 1: heat1d benchmark in input formats for different tools

is to use two nested loops. Figure 1a shows the same algorithm written in C formatted for the tool Fluctuat [13]. A loop representation is more succinct—14 lines of code with computations, however it requires loop bounds to be set manually and may lead to index-out-of-bounds errors.

We show the same function *heat1d* written in our functional DSL in Figure 1b. It uses a sliding window over a list (`slideReduce` operation, explained in more detail in Section 3.3) and passes the new values into a recursive call. In contrast to alternative implementations, a functional style program is much shorter—6 lines of code—and eliminates index-out-of-bounds errors as it does not require users to explicitly write elements' indices.

DSL Design Our DSL is designed for writing numerical algorithms on array-like data structures and was inspired by the popular libraries Lift [15] and TensorFlow [23]. It includes commonly occurring operations on vectors and matrices from the collected benchmarks. When naming DSL functions, we have re-used the names used by Lift and TensorFlow whenever possible and attempted to make other functions' names self-explanatory. We do not expect our current DSL to exhaustively cover all possible numerical programs; rather it serves as a starting point already covering a variety of operations that can and should be extended in the future.

Data Types Following previous work in rounding error analysis, all values and operations in our DSL are real-valued (as opposed to finite precision), i.e. they have a `Real` type [4]Real-valued algorithms are more intuitive for a user to write, and easier to analyze as they provide a clear reference semantics. Our DSL provides two data types: a `Vector` is an indexed sequence of `Real` scalar values, and a `Matrix` corresponds to a sequence of vectors of the same length. In the following, we refer to lists (`Vectors`) as vectors, and vectors and matrices as data structures (DSs), for simplicity. Our DSL is purely functional, and as such all data structures are immutable.

To analyze a real-valued program, a user should specify the finite precision, for which the rounding error will be computed. By default our analysis computes the error for a uniform double floating-point precision. Alternative precision assignments can be passed as an additional parameter to our tool and are not a part of the DSL itself.

Input Ranges Any rounding error analysis requires information on ranges of input variables. Both scalar and DS input ranges can be specified using the `require` clause. The specification should ideally be as precise as possible and provide tight ranges that can be different for some DS elements. We therefore allow two ways to specify input ranges for DSs. If all elements have the same input range, it is enough to specify the range once for the whole DS (`1.0 <= ax && ax <= 2.0`). Additionally, it is possible to specify individual input ranges for subsets of DS elements. For vector elements these ranges are specified as a tuple $((loInd, hiInd), range)$, where *loInd* and *hiInd* are the smallest and the largest index of consecutive elements with the input range *range*. For example, to specify that the first and the second element of `ax` in *heat1d* have the input range $[0.0, 0.5]$, we would write `ax.range(0, 1)(0.0, 0.5)`. We also allow individual range specifications on matrices, however, specifying a lower and upper bound of an index range is ambiguous for a matrix. Therefore, we choose a more natural way for specifying special input ranges on matrices: a user has to list the indices of elements for that range. For example, to convey that the first element in the first and second row of a matrix `m` should have the range $[-0.5, 0.5]$, we write `m.specM(Set(Set((0,0),(1,0)),(-0.5,0.5)))`.[5]

DS Size To analyze operations that traverse a DS, the analysis also needs to know the number of elements in the DS. Our DSL allows to specify the expected *maximum size* of an input data structure—length of a vector, number of rows and columns for a matrix. Having the upper bound on the number of elements in the DS allows us to compute sound results: reported ranges and rounding errors subsume the ranges and errors of programs with input DSs smaller than the specified size.

[4] Precisely, all *fractional* numbers have the type `Real`, while DS sizes and indices have the integer type `Int`.

[5] Admittedly, the `Set()` notation is not the most user-friendly way of input for small specifications. We use it for simplicity of implementation; the notation can be improved with extensions to our parser.

3.3 DSL Functions

Our DSL uses Scala syntax (a representative subset is available in the appendix in Figure 5); semantically we can roughly split its functions into four groups:

1. *element-wise functions*, such as arithmetic operations and transcendental functions applied to individual elements of a DS;
2. *standard functions*, such as `map`, `fold` and `filter`;
3. *domain-specific* functions, e.g., stencil-like filters, matrix multiplication;
4. *non-numerical* operations, e.g., appending or flipping elements in DS.

Additionally, our DSL supports recursive calls with specific conditional statements. To avoid rounding errors in conditional expressions, we currently limit them to (integer) DS size comparisons `dssize` $\leq c$. Since we only handle *bounded* loops, the DS size must be finite and decreasing in each recursive iteration. Next, we explain the concrete semantics of the DSL functions using pseudocode that makes indices explicit (while they are typically implicit in our DSL). We choose to present the semantics with pseudocode (and not sets of rules), because it is more concise and because it expresses how the operators are ultimately evaluated, which is important for the rounding error analysis.

The semantics of most of our DSL operators is standard. Additionally, our analysis does *not* depend on exactly this DSL's syntax and semantics. We therefore expect our analysis to be *applicable to other (intermediate) representations or languages* with similar semantics. Such representation must (only) be purely functional (immutable variables and DS, no side-effects) and provide a syntactic distinction between different iterators, precisely, the functionality of an iterator must be unambiguous without an additional analysis of the iterator's body.

Element-wise Functions They cover arithmetic operations applied to a single DS or a pair of DS, for instance $v1 + v2$, where $v1, v2$ are vectors. Semantically these operations are the same as arithmetic operations on scalar numbers. The only difference is that for binary operations on two DS, the operands must have the same dimension. Element-wise operations are defined for both vectors and matrices: the operation is applied to the elements in the operand DSs with the same indices. We also define element-wise operations with constants.

For all unary (`uop`) and binary (`bop`) arithmetic operations the semantics is:

```
a bop b = [a[i] bop b[i] | ∀i∈Indices(a), #Indices(a) == #Indices(b)]
uop(a) = [uop(a[i]) | ∀i∈Indices(a)]
```

In our example function *heat1d* in Figure 1b (line 8) the expression `coef*v` is an element-wise multiplication of vectors `coef` and `v`: the i-th element of the output vector contains the result of multiplying the i-th element of `coef` with the i-th element of `v`.

Standard Functions Classic functional-style functions `map, fold, filter` preserve their semantics. `map` and `fold` are defined on vector elements, and for a matrix on both rows and elements. We add a function `ds.sum()` as syntactic

sugar for `fold` with an addition operator to compute a sum of DS elements. We also extend the map on matrix rows to support indexed iterations with `enumRowsMap(`λ`i,x.`f`(i,x))`. The function maps over rows of the matrix and applies f to both row's index and elements:

```
m.enumRowsMap(f) = [f(i, m[i,j]) | ∀i∈Rows(m), (i,j)∈Indices(m)]
```

`filter` is defined to apply the conditional to vector elements, and to matrix rows. We do not allow a `filter` on individual matrix elements, as it may result in modified and uneven matrix dimensions.

Domain-Specific Functions Our DSL defines operations required for implementing neural networks (i.e. matrix multiplication), stencils and image processing filters. We describe the most interesting operations below.

Stencil operations usually require a more complex transformation than `map` or `fold` can provide. The transformations involve an outlook of several elements before and after the current element of a DS, as opposed to accessing a single element in one iteration of `map` and `fold`. Such an outlook is commonly called a sliding window. Our DSL defines it on vectors and matrices with `ds.slideReduce(size, step)(`λ`x.`f`(x))`, where a window of size `size` shifts by `step` indices at every iteration. For vectors a window is a subset of consecutive elements of length `size`, for matrices a window is a matrix with dimensions `size`$\times$`size`. A user-supplied function $f(x)$ is then applied to the created window, it returns a scalar value that is saved at the corresponding index of the newly created DS. Intuitively, it is similar to applying a `fold` to a sliding window.

Our example benchmark *heat1d* in Figure 1b creates a sliding window of 3 vector elements and shifts the window by 1 index at every iteration; the resulting vector `updCoefs` contains results of the `sum()` operation. The pseudocode below explains `ax.slideReduce(3,1)(f)` using explicit indices of the vector `ax`:

```
k=0
∀ i∈ {1..size(ax)-2}:
   v = [ ax[i-1], ax[i], ax[i+1] ]
   updCoefs[k] = coef[0]*v[0] + coef[1]*v[1] + coef[2]*v[2]  // f(v) = (coef*v).sum()
   k++
```

The pseudocode contains two indices: i is the index of elements in the original DS `ax`, and k is the index of a sliding window over `ax` and the output vector `updCoefs`.

Our DSL also allows a combination of a sliding window and a `map`, which is useful for implementing signal filters such as the fast Fourier transform. The function `enumSlideFlatMap(n)(`λ`i,x.`f`(i,x))`, defined on vectors, creates a sliding window of size n that shifts by n indices every iteration. The resulting windows are enumerated and a function $f(i, x)$ transforms every element in the window and saves the results into a new vector. In the FFT implementation in Figure 2, the sliding window includes 2 elements of the vector `evens` and computes vectors `resleft` and `resright` of the same size as `evens`. The window index `k` is used for accessing elements of the vector `odds` and for computing the filtered values (lines 16 and 22). The pseudocode below explains with explicit indices how `evens.enumSlideFlatMap(2)(f(k,xv))` iterates over the vector `evens`:

```
1  def fftvector(vr: Vector, vi: Vector): Vector = {
   // v: (real part of signal / Fourier coeff.,
3  // imaginary part of signal / Fourier coeff.)
   require(vr >= 68.9 && vr <= 160.43 && vr.size(128) &&
5          vi >= -133.21 && vi <= 723.11 && vi.size(128))
    if (vr.length() == 1)
7     Vector(List(vr.head, vi.head))
    else {
9     val scalar: Real = 1; val Pi: Real = 3.1415926
      val n:Int = vr.length(); val direction:Vector = Vector(List(0.0, -2.0))
11    val evens: Vector = fftvector(vr.everyNth(2, 0), vi.everyNth(2, 0))
      val odds:  Vector = fftvector(vr.everyNth(2, 1), vi.everyNth(2, 1))
13    val resleft: Vector = evens.enumSlideFlatMap(2)((k, xv) => {
        val base: Vector = xv / scalar
15      val oddV: Vector = odds.slice(2 * k, 2 * k + 1)
        val expV: Vector = (direction.*(Pi * k / n)).exp()
17      val offset: Vector = (oddV x expV) / scalar
        base + offset })
19    val resright: Vector = evens.enumSlideFlatMap(2)((k, xv) => {
        val base: Vector = xv / scalar
21      val oddV: Vector = odds.slice(2 * k, 2 * k + 1)
        val expV: Vector = (direction.*(Pi * k / n)).exp()
23      val offset: Vector = (oddV x expV) / scalar
        base - offset })
25    resleft ++ resright  })
```

Fig. 2: Fast Fourier transform filter implemented in our DSL

```
k=0
∀ i∈ {0,2,4,...,size(evens)-2}:
   xv = [ evens[i], evens[i+1] ]
   tmp = f(k, xv) // where tmp is a vector
   res[i] = tmp[0]; res[i+1] = tmp[1]
   k++
```

Non-Numerical Such operations include obtaining a subset of elements (`v.slice(i,j)`), reordering (`m.flipud()`, `m.fliplr()`), appending and prepending elements and rows (`v.+:(elt)`, `m :+ v`). Additionally, our DSL allows to add a zero-padding around a vector or a matrix, and obtain smallest and largest elements of a DS. A special variant of a subset operation `ds.everyNth(n, fromInd)` creates a new DS by taking every n-th element of a vector (or row of a matrix) starting from the index *fromInd*. Our FFT benchmark in Figure 2 uses the `everyNth` function to obtain subsets of signal values at even and odd indices (lines 11 and 12).

4 Data-Structure Guided Analysis

While a baseline range and error analysis for straight-line code can handle unrolled iterators, it does not make use of implicit additional information that is present in a high-level specification. In an unrolled program each iteration makes up independent expressions to be evaluated, regardless of whether values in consecutive iterations depend on one another. This may result in redundant computations; for instance, a `map` performs the same computation over all elements in a vector and when all those elements have the same specified input range, we only need to analyze the rounding error of the computation once. The same

holds for matrix multiplication: each element of the resulting matrix is computed with the same arithmetic expression, but it would appear as a new independent computation if unrolled. When sets of involved elements have the same ranges, it is sufficient to analyze the rounding error of the resulting matrix element once.

We observe that while concrete DS inputs will in general not be the same, a specification of a function to be analyzed will typically provide ranges that in practice often tend to be identical for many inputs. We leverage this in our analysis and compute the ranges and error bounds as rarely as possible. Even though it is not possible to directly apply this approach to iterators where iteration values have dependencies, like `fold`, the analysis can be optimized based on groups of elements with the same specification by introducing suitable over-approximations (see Section 4.4).

We first introduce our DS-based concrete and abstract domains before explaining how expressions are analyzed and their analysis is optimized.

4.1 DS-based Concrete Domain

The goal of our analysis is to collect information about ranges and rounding error bounds for groups of elements. To do so, our concrete domain tracks a tuple (r, f) for each value in a program, where r is the ideal value if a program would be executed with a real numbers semantics, and f is the same value if the program is executed with the finite-precision semantics.

We denote all valid indices of data structures as $Inds^{(n)} = \mathbb{N}^n$, where $n \geq 0$ is the dimension of the DS: $n = 1$ for vectors and $n = 2$ for matrices. For scalar values the set of indices is empty, $n = 0$. Using the indices we define elements of a DS as $\mathbb{V}^{(n)} = Inds^{(n)} \mapsto (\mathbb{R}, \mathbb{F})$. Given a set of elements' values $\mathbb{V}^{(n)}$ we define our concrete domain as $\mathbb{C}^{(n)} = 2^{\mathbb{V}^{(n)}}$, for each dimension of data structures n.

4.2 DS-based Abstract Domain

We then abstract each tuple (r, f) using a pair of real-valued intervals: $\alpha((r, f)) = (I_R \times I_R)$, where the first interval denotes a range of real values that contains r, and the second tightly bounds the difference between r and f. Here the difference between a real number r and a finite-precision number f represents the rounding error.

Lifted to the DS with dimension n we obtain abstract element's values: $\mathbb{D}^{(n)} = Inds^{(n)} \hookrightarrow (\mathbb{I}_R \times \mathbb{I}_R)$. Note that we are only interested in abstract values of elements with valid indices (as opposed to all possible indices), and use a partial mapping $\hookrightarrow$ to express it in our domain. For invalid indices the mapping is undefined. The abstract domain for our analysis combines all $\mathbb{D}^{(n)}$ with for scalar values, vectors and matrices: $\mathbb{D} = (\mathbb{D}^{(n)})_{n \geq 0}$. Join and meet operators use standard definitions of join and meet on intervals, and are lifted to all valid indices point-wise.

An abstract state $D^{(n)}$ soundly describes a concrete state $C^{(n)}$, that is: $C^{(n)} \subseteq \gamma(D^{(n)})$, where concretization function is defined as follows. Given a set

of indices S and a set of mappings from these indices $D^{(n)} = \{i \mapsto (I_i, E_i) | i \in S\}$:

$$\gamma(D^{(n)}) = \{\{i \mapsto (r_j, f_j) | i \in S\} | \forall j. r_j \in I_i, |r_j - f_j| \in E_i\} \tag{2}$$

The abstraction function α is defined as an adjoint of γ: $\alpha(c) = \bigsqcap\{a \mid c \in \gamma(a)\}$, so they form a Galois connection. Each transformation of the abstract state is parametrized with an expression to be evaluated, a mapping of variables' values and computes a new abstract state:

$$[\![\cdot]\!]^\sharp = Expr^{(n)} \to (Vars \overset{\text{n}}{\mapsto} \mathbb{D}^*) \to \mathbb{D}^{(n)}, \tag{3}$$

where $\overset{\text{n}}{\mapsto} \mathbb{D}^*$ is a type-preserving mapping that assigns $D^{(0)}$ values to scalar variables, and $D^{(1)}$, $D^{(2)}$ values to vector and matrix literals respectively.

Theorem 1. ***Soundness.*** *Given an abstract state $D \in \mathbb{D}^{(n)}, \{i \mapsto (R, E)\} \in D$ there exists no concrete state $C \in \mathbb{C}^{(n)}$ such that $C \subseteq \gamma(D)$, $\{i \mapsto (r, f)\} \in C$ and $r \notin R \vee |r - f| \notin E$. Moreover, if $D \in \alpha(C)$, $[\![e]\!]C = C'$, and $[\![e]\!]^\sharp D = D'$, then $D' \in \alpha(C')$.*

Proof. (sketch) The theorem states that there is no unsound abstract state in our analysis, and given a sound starting state, our abstract transformations result in a sound end state. The first part follows directly from the definition of interval abstraction and concretization.

The transformations $[\![\cdot]\!]^\sharp$ on data structures are defined for each individual element, which reduces them to transformations on basic blocks. The conditional expressions allowed in the language do not introduce instabilities or discontinuity errors [8,31] and thus do not require special treatment. Precisely, our DSL allows only two types of conditionals: 1) an *integer* comparison of the DS size with a constant in the recursive call, and 2) a comparison with a constant $x \leq c$ inside the `filter` function. As explained later in more detail, our analysis overapproximates the results of `filter` by keeping all DS elements that *may satisfy* the condition. Hence, the result of $[\![\texttt{filter}]\!]^\sharp$ is at least as large as the resulting DS size in concrete semantics, while the DS elements themselves remain unchanged. The result is consistent with the semantics of the input DS size specification and can be used by further iterators over the "filtered" DS. As the conditionals do not introduce instability and all iterators can be unrolled, soundness of our analysis follows from the soundness of the underlying baseline analysis for straight-line code. □

Our functional DSL defines all DS to be immutable, therefore each element of a DS is only assigned once. Our abstract domain does not require updates to individual element's ranges, and all recursive calls are unrolled. Since our analysis handles *only bounded loops* by design, we can unroll all operations, if needed, which is why we do not provide an additional widening operator. While widening in general allows the analysis to terminate quickly, for rounding error analysis the performance/accuracy trade-off is too costly. As our experiments with Fluctuat show (Section 6.1), for rounding error bounds, precision lost with widening cannot be recovered, hence an analyser that uses widening in the vast majority of cases reports infinite error bounds, which is sound but not especially meaningful.

4.3 DS Analysis

Both concrete and abstract domains partition DS elements in groups based on their real value and value range respectively. Our implementation describes a group of elements using a *set of indices*. The indices in one group need not be consecutive, the only condition is that they correspond to unique and valid indices of DS elements. Thus when analyzing an operator such as `map`, adding or multiplying by a constant, we only need to run the analysis once per group.

The initial grouping of elements is defined by user range specifications on the input DS. For intermediate variables in the computations, numerical indices for an abstraction of DS elements are inferred during the analysis. Note that the grouping does not change the semantics of functions and operators. As our DSL operates on real numbers, for commutative operations on DS elements their order does not matter. Whenever the analysis encounters an operation where the order of elements does matter, e.g. when computing an accumulator value in `fold`, we sort and split the groups to only contain consecutive elements' indices.

Whenever the expression under analysis contains only scalar values and operations, our analysis re-uses the baseline dataflow rounding error analysis, described in Section 2. We next describe how our analysis handles different kinds of DS operations.

Example We illustrate our abstraction using the running example program in Figure 3. This contrived example is not part of our benchmark set, we use it here purely for demonstrating the relevant DSL details in a succinct way. Function `fun` takes two input vectors `x` and `y`, both of size 5. An abstraction for vector `x` keeps track of separate ranges for the first two elements and the remaining ones (with indices 2,3,4), i.e. $D_x^{(1)} = \{\{0,1\} \mapsto [0.5, 1.5], \{2,3,4\} \mapsto [0, 10]\}$. For the input vector `y` the abstraction also has two groups, but indices in the first group are not consecutive: $D_y^{(1)} = \{\{0,4\} \mapsto [-1, 2], \{1,2,3\} \mapsto [0, 1.5]\}$.

Map and Element-Wise Operations Our domains group elements that have the same real range by their indices, such that we can perform range evaluation once for each group. The most prominent example where such evaluation makes a difference for performance is the `map` function, such as on line 4 in Figure 3. The program multiplies all the elements of the list resulting from `x + y` by 2.0 and adds 1.5. The individual multiplications and additions are independent of each other, i.e. they do not propagate through iterations. For DS elements in one group we thus evaluate the range and error of `i*2.0 + 1.5` only once.

We use a similar approach for element-wise arithmetic operations between two vectors (or two matrices), such as `x + y` in Figure 3. In contrast to `map`, element-wise operations are binary and we need to take into account *pairs* of ranges. For each unique pair of ranges of operands we compute the range (and error) once.

Matrix Multiplication Evaluation of matrix multiplication is similar to the element-wise operations, where we compute pairs of ranges. Except, for matrix multiplication the elements, for which we need to know the ranges are located at the left-hand-side matrix row and the right-hand-side matrix column. We con-

```
def fun(x: Vector, y: Vector): Real = {
 require(x>=0.0 && x<=10.0 && x.size(5) && x.range(0,1)(0.5,1.5) &&
        y>= -1.0 && y<=2.0 && y.size(5) && y.range(1,3)(0.0,1.5))
   val z = (x + y).map(i => i*2.0 + 1.5)
   val r = z.fold(1.0)((acc: Real, i: Real) => acc * sqrt(i))
   r / (x.length()) }
```

Fig. 3: Example program in our DSL

struct an expression for computing the resulting matrix elements internally. For each unique pair of ranges we only evaluate this expression once.

Filter Filter also takes advantage of the element grouping; our analysis evaluates the condition on each group of DS elements only once. However, `filter` is different from the rest of the functions in our DSL, because its abstract semantics do not exactly mirror the concrete. In the concrete semantics, `ds.filter(`$\lambda x.f(x)$`)` partitions the DS `ds` into two disjoint sets: elements that satisfy `f(x)`, and that satisfy its negation. In the abstract semantics these sets are not necessarily disjoint. Our evaluation `eval` returns an over-approximation of a set of elements from `ds`: the elements that *may* satisfy the condition `f(x)`. Currently we limit expressions in `f(x)` to simple comparisons $x \leq c$ and $x \geq c$, where `x` is the DS element and `c` is a scalar variable or a constant. More complex arithmetic operations are likely to introduce rounding error inside the condition itself, which may lead to a discontinuity error—elements that would have satisfied `f(x)` in a real-valued expression, do not satisfy it under floating-point semantics (or vice versa). We note that complementary techniques for bounding this discontinuity error [8,31] exist that may be integrated into our analysis.

Unrolled Operations Naturally, not all operations can benefit from a grouping of DS elements alone. The "once-per-range" evaluation cannot be applied on operations that propagate values through multiple iterations (`fold, slideReduce`) or use fresh values at each iteration (for example, loop counters in `enumSlideFlatMap, enumRowsMap`). For these functions, the abstraction-guided analysis falls back to the baseline version. It unrolls the iterators and performs range and error evaluation once for each iteration, we then join the ranges (for values and, separately, for errors) to ensure that our results subsume all evaluated iterations. Our analysis handles recursive calls in the same way and unrolls each call as one iteration. Note that for our analysis to terminate, a recursive function must contain an exit condition that uses the (decreasing) length of a DS.
In our running example the analysis unrolls `z.fold` and evaluates ranges and errors of the unrolled expression:
`1.0 * sqrt(z.at(0)) * sqrt(z.at(1)) * sqrt(z.at(2)) * sqrt(z.at(3)) * sqrt(z.at(4)).`

Non-Numerical Operations Operations that do not involve arithmetic computations do not introduce new errors, however, they do affect our abstraction. For example, a prepend operation `x.+:(8.0)` will add an element with index 0 and range $[8, 8]$ to the abstraction and shift all indices of `x` by one. If we apply `x.+:(8.0)` to the `x` in the running example, the resulting abstraction will become

$D_x^{(1)} = \{\{0\} \mapsto [8,8], \{1,2\} \mapsto [0.5,1.5], \{3,4,5\} \mapsto [0,10]\}$. Similarly, the `pad` operation adds elements with range $[0,0]$ around a vector or matrix and re-scales the original elements' indices. Another interesting case of the non-numerical operations is the `x.everyNth(n,k)` function that constructs a new DS by appending every n-th vector element (or every n-th matrix row) starting from the index k and assigning new indices to them. Evaluating `x.everyNth(2,0)` on the $D_x^{(1)}$ from our running example will result in $D_{nth}^{(1)} = \{\{0\} \mapsto [0.5,1.5], \{1,2\} \mapsto [0,10]\}$.

4.4 Optimized Evaluation of `fold`

The `fold` function cannot be evaluated only once per range group, since the accumulator's value changes at every iteration. For analysis, it would thus have to be unrolled. We observed, however, that in many applications the function passed to `fold` has a rather simple structure, such as summing up all elements of the DS. For such simple iterator bodies, the explicit unrolling can be replaced with an optimized evaluation that benefits from grouping of elements.

Our optimization over-approximates the accumulator, thereby effectively eliminating the change in input values from iteration to iteration. The analysis then computes one range per group of elements using a closed-form formula. In general, it is also possible to use approximation of an accumulator and a DS element for the whole loop, not only per group of elements with the same range. However, such a computation will introduce an even larger over-approximation in the result. To keep the bounds reasonably tight, we choose to apply over-approximations rarely. We have implemented this optimization for the most common special cases of lambda functions `f()` that follow next.

Linear loop In a linear loop, i.e. $f(ac, el) = a \cdot el + b \cdot ac + c$, if f is executed on a group of elements with the same range $range(el)$, then we can compute the resulting range after n iterations with:

$$range_n = a \cdot range(el) \cdot \sum_{i=0}^{n-1}(b^i) + b^n \cdot init + c \cdot \sum_{i=0}^{n-1}(b^i), \tag{4}$$

where $init$ is the initial value of the accumulator for the current group of elements. The initial accumulator value changes from group to group: it starts with the input parameter of `fold` and for each consecutive group it is replaced with the result of the previous computation. To account for all combinations of signs of linear coefficients a,b,c, we take their ranges to be symmetrical around zero. For generic linear loops, the order of computations matters, therefore we sort and split the groups in the abstraction $D^{(n)}$, such that each group only contains elements with consecutive indices, and the computation is applied to each group in the natural order: starting with the group containing index 0.

There is no simple closed-form equation to compute the rounding errors for linear loops. We therefore unroll the loop for error computations, but we use the over-approximated range of `acc`, pre-computed using Equation 4. Note that such an evaluation is faster than the full unrolling, since we pre-compute the ranges necessary for error computations.

Sum A sum of all elements in a vector or matrix is a special case of a linear loop, but in the absence of linear coefficients the range computations are much simpler. For a function $f(acc, el) = acc + el$, we compute one range per group of elements in $D^{(n)}$ abstraction using the formula: $n \cdot range(el) + init$, where n is the number of elements in the group, and $init$ is the initial value of the accumulator for the current group. Note that here the order of groups does not matter, as our DSL specifies a program over real numbers and real-valued sum is associative. The error computation is performed similar to linear loops: we over-approximate the value of `acc` and use the range to compute the error on the unrolled `fold`.

5 Implementation

We implement our analysis in a tool called DS2L in the Scala programming language. For performance reasons, we implement all internal computations using intervals with arbitrary-precision bounds (with outwards rounding for soundness), using the MPFR library [12] with 128 bits of precision. We use the intervals for both range and error computation, and sacrifice some of the error accuracy compared to affine arithmetic that is used by most state-of-the-art analyzers.

We choose to implement the partitioning using sets of indices, among other alternative representations: linear inequalities [3,16], difference-bound matrices [3], and sets of other simple symbolic expressions [5]. We choose a set representation because it does not depend on patterns to group the elements. We have empirically confirmed that on our benchmarks the set representation of index groups performs better than symbolic ranges of consecutive indices. This is becuase our range evaluation often needs to obtain the range of a DS element with a given index[6], which is a simple inclusion check for sets, but requires additional computation of numerical bounds from symbolic expressions in other representations.

In this paper, we consider only the natural order of evaluation (left-to-right with call-by-value), exactly as it syntactically appears in the program under analysis. For this natural order, DS2L generates executable Scala code and for that code the analysis is sound. Our analysis can also be adapted to other, more efficient, evaluation orders, but determining that order is an orthogonal issue.

6 Experimental Evaluation

We evaluate our DS-based analysis in DS2L in terms of performance and accuracy, focusing on the following research questions:

RQ1 How does DS2L compare to state-of-the-art tools (on large programs)?
RQ2 How does DS-based abstraction affect the accuracy/performance tradeoff?

Benchmarks We evaluate DS2L on the new benchmark set we collected (Section 3.1). The original codes were written in different programming languages.

[6] For instance, taking a single element's range or a range of a group of elements when unrolling an iterator.

Table 1: Benchmarks description: usage of DSL functions and unrolled program sizes for different DS size configurations (in lines of code)

Benchmark	DSL usage						max #ops	Benchmark sizes		
	map	fold	slideRed.	enum*	rec	matMul	in line	small	medium	large
vector benchmarks										
avg		✓					1	101	1001	10001
variance		✓					3	202	2002	20002
stdDev.		✓					3	202	2002	20002
roux		✓					3	100	1k	10k
goubalt		✓					3	100	1k	10k
harmonic		✓					3	200	2k	20k
nonlin1		✓					7	200	2k	20k
nonlin2		✓					8	200	2k	20k
nonlin3		✓					6	200	2k	20k
heat1d	✓		✓		✓		5	257	1025	65537
fftvector	✓	✓		✓	✓	✓	4	96	9596	48636
matrix benchmarks										
pendulum		✓					4	404	4004	40004
alphaBlend.	✓						4	100	1k	250k
fftmatrix	✓	✓		✓	✓		8	64	6012	30204
conv.2d_sz3			✓				1	162	1458	118098
sobel3			✓				3	972	8748	708588
lorentz		✓					6	141	211	281
lyapunov	✓					✓	(20,200,1000)†	11	101	501
control.Tora	✓					✓	(20,200,1000)†	31	301	1501

† The benchmark contains matrix multiplication, the maximum number of arithmetic operations in one line of code depends on the size of multiplied matrices. Reported values are for (small,medium,large) input DSs.

We have translated them into our functional-style DSL for the purpose of our evaluation and validated our translation with testing. Table 1 displays in more detail which elements of our DSL were used in which benchmarks. Many of the benchmarks operating on vectors have been repurposed from controller loops used in previous work [6] and therefore have similar structure. As an artifact of this translation, our vector-based benchmarks use `fold` frequently.

For each benchmark, we create 12 variants by varying the following:
Size of the input DS. Input vectors are assigned 100(small), 1k (medium), or 10k (large) elements. Input matrix sizes are 10x10 (small), 100x100 (medium) and 500x500 (large). For benchmarks where the size of a DS is predefined by the algorithm, we take the sizes closest to 10, 100 and 500 (for example, the input matrix for *fftmatrix* has 8x2, 128x2 and 512x2 elements for the small, medium and large setting, respectively). The benchmark input DS size influences the number of operations to be evaluated by the analysis. To give an unambiguous measure of complexity of the programs under analysis, we report the sizes of unrolled programs in Table 1. The reported numbers are lines of code if all operations on DSs would be unrolled to scalar operations, i.e. the number of iterations times number of lines of code computing a scalar value inside each

iterator. Since in the absence of DSs there would be no need for non-numerical functions as concatenation of vectors or changing the order of elements in a matrix, we only count lines of code with numerical operations and let-statements. Such unrolled programs could, for example, be used by state-of-the-art rounding error analyzers that operate on straight-line code. Additionally we report the maximum number of arithmetic operations in one line of unrolled code.

Our goal is to efficiently analyze *large* benchmarks. We include small and medium sizes for completeness and to demonstrate scalability, but do not consider DS2L to be necessarily the analysis tool of choice for these.

Range specification granularity. We vary the amount of individually specified ranges per DS. The input ranges are specified with either one, i.e. the same, interval for all elements (*AllSame*), different intervals for all elements (*AllDiff*), or for some. When specifying individual ranges for subsets of elements we vary the amount of new range specifications to be 10% and 30% of the input DS size (*Diff10P* and *Diff30P*). For instance, if an input vector has 100 elements *Diff10P* configuration will have 10 additional range specifications, each with an arbitrary amount of elements in it, and the *Diff30P* will have 30 additional range specifications. To avoid any bias by using input ranges that are easier for the analyzer to compute with, we generate all input ranges randomly. Similarly, the amount of elements in one group with special ranges is determined randomly. The smaller ranges of more refined specifications are subsumed by the ranges in AllSame specification.

Experimental Setup To answer our research questions we evaluate differences in accuracy and performance between a baseline analysis, our new DS abstraction-guided analysis and state-of-the-art tools. To do so, we normalize the reported worst-case rounding error and the running time of the analysis (separately) with respect to a baseline (different for each comparison). Such a normalization is necessary since the running times and error magnitudes vary widely between different benchmarks due to their diverse complexity. We then evaluate the normalized worst-case errors and analysis times.

As running time, we use the reported analysis time of each tool. This is a subset of the total wall-clock running time and excludes, for instance, parsing of the input programs. Since the formats of the input programs differ widely, we consider the analysis time a more meaningful measure for a comparison. We report analysis time averaged over 3 runs. We consider that a tool failed on a benchmark if it either timed out with 30 minutes, reported an infinite error bound, or encountered some other error. Timeouts were always consistent across all runs on each configuration. Note that the timeout applies to the total running time, including parsing, pre- and post-processing of the results.

As accuracy measure, we use reported absolute worst-case rounding error bounds of each tool for double floating-point precision. For the 13 benchmarks where the return type is a vector or a matrix we take the maximum error of all output DS elements.

All experiments were run on an Intel Xeon machine with 8 CPUs @ 3.50GHz, 32G of RAM under the OS Ubuntu 22.04. We run both DS2L and a baseline straight-line code analysis in a JVM with 2G memory and 1G stack space.

6.1 State-of-the-Art Tools

We compare DS2L against the state-of-the-art rounding error analyzers Fluctuat [13] and Satire [9]. We choose these two tools, because they are the only tools that natively support data structures and loops over them (Fluctuat), or that analyze straight-line code, but whose abstractions were designed specifically for large program sizes (Satire). In these two dimensions that are relevant for our comparison, Fluctuat and Satire are the state-of-the-art. Satire does include approximations such as not considering higher-order terms that technically affect its soundness, but we ignore this here. DS2L and Fluctuat are 'fully sound'.

We note that an entirely fair comparison is not possible due to the different input formats, as well as different implementation choices such as programming language in which the tools themselves are implemented. Each of our high-level benchmarks written in our functional DSL can be translated to Fluctuat's and Satire's imperative formats in different ways that each may or may not affect the results (no guidelines exist). We manually translate our benchmarks into the tool's input formats by choosing the way that we consider to be natural for a programmer, and so a regular user of the tools would choose, and validate the translation with testing.

In our comparison, we use relative performance and accuracy as a measure of success. DS2L and Fluctuat are deterministic and always report the same error bounds. On some benchmarks Satire reported slightly different errors, we take the largest reported error across the runs. Note that the differences were on the order of 10^{-12}, and taking the average or the smallest error across the runs does not affect the qualitative results.

Fluctuat Fluctuat can both unroll loops internally and abstract the loop behavior by applying widening. We use the latest available version of Fluctuat provided to us in October 2022.

Fluctuat takes C-programs as input and is itself implemented in C. When translating our benchmarks, we tried to preserve as much functional-style semantics as possible, but had to give up the DS immutability and replace all recursive calls by loops. Furthermore, Fluctuat's library did not support a `max()` function required for implementing the ReLU function in the neural network benchmarks *lyapunov* and *contr.Tora*. We replaced the call to `max()` with an explicit if-then-else statement. Fluctuat does not have a dedicated way of specifying input ranges for data structures, only for scalar values. We therefore assign a range to each element separately, and use loops to assign repeating ranges for the *AllSame* specification. Each benchmark is implemented in a separate function that is called from `main`. We compare DS2L with Fluctuat on all 19 benchmarks.

We run Fluctuat with several different settings:

1. loop iterations are evaluated separately, results joined (merge over all paths—MOP—solution)
2. loops are unrolled until 50k iterations. The largest number of iterations in our benchmarks is 62.5k, however, Fluctuat's setting did not allow us to set the unroll limit higher than 50k.
3. loops are abstracted by widening, nothing is unrolled
4. automatic setting, where Fluctuat finds a suitable number of loop unrollings before applying joins and widening.

Out of all configurations the overall best results were achieved with MOP (which is effectively unrolling) and the explicit unrolling configuration. Fluctuat with MOP and unrolling has timed out less often than other configurations and whenever Flucutat computed non-trivial error bounds, they were exactly the same for all settings. Surprisingly, the automatic configuration of Fluctuat had the highest timeout rate: it failed to produce results within 30 minutes on 33% of specifications. The pure widening configuration performed better with only 16% rate of timeouts. Since all other settings provided worse or the same results, we compare DS2L's results only to the MOP setting of Fluctuat.

Satire We use the latest version of Satire available in the open-source GitHub repository[7]. Satire's open-source benchmark set contains pre-processed large unrolled loops, but no original programs that were unrolled. Unfortunately, the original programs with loops were not available (upon request). We have therefore reverse-engineered the loops over data structures from their unrolled versions for two benchmarks *lorenz*, and *heat1d*. Additionally, we translated some of our benchmarks into Satire's input format, which is an imperative DSL that specifies floating-point precision for each variable assignment. We only compare the results on a subset of benchmarks, since we are required to manually unroll the loops, and translate functional operators into imperative code. This translation process is non-trivial, tedious and error prone, especially for complex functions.

Overall, we translated 9 benchmarks that contain a `fold` over an input vector. For these 9 benchmarks we used the same variations in configurations, described above: small, medium, large input DS sizes, and *AllSame, Diff10P, Diff30P, AllDiff* specification granularities. We took Satire's original benchmarks as is: *heat1d* had only one version, that corresponds to our input specification with small input DS and one input range for all elements. The *lorentz* benchmark was available in three different sizes of input DS (20, 30 and 40), and all of them had the same input range for all elements of DS (*AllSame*). In total, we have compared our results on 112 benchmark variations.

We ran Satire with its default parameters and both with and without abstraction. The version with abstraction predictably produced results faster and had fewer timeouts. We therefore compare to the version of Satire with abstraction enabled.

[7] We use the version with the commit hash 8a4816aac6fad4fb86c2af8dc8e634bf02912b90.

Table 2: Relative accuracy/performance of state-of-the-art tools compared to DS2L with DS abstraction.

Benchmark size	Accuracy			Performance			# fails	# fails DS2L	total # of bench.
	min	median	max	min	median	max			
Fluctuat									
Small	2.11e-07	0.557	3.55	0.07	0.36	7.22	**2**	10	76
Medium	2.83e-04	0.639	2.91	0.23	**1.98**	4636.52	22	**12**	76
Large	3.60e-02	0.555	2.91	0.66	**24.41**	339.55	60	**25**	76
Satire									
Small	2.98e-07	0.737	3.54	6.33	**24.68**	449.33	6	6	38
Medium	2.07e-10	0.153	3.54	6.32	**39.90**	507.45	12	**8**	37
Large	3.94e-02	0.953	1.34	8.95	**259.64**	767.13	32	**10**	37

6.2 RQ1: Comparison to State-of-the-Art Tools

We compare relative performance and accuracy of state-of-the-art tools normalized against DS2L's results and provide cumulative values in Table 2. The values greater than 1 denote individual benchmarks where DS2L was faster (respectively, more accurate) than the state-of-the-art tool. For instance, value 24.41 means that DS2L is median 24.41 faster than Fluctuat. As it is ambiguous to compute the relative value if one of the tools did not report results, we do not include these cases into the minimum, median and maximum values. Instead we report the number of failures per tool (timeouts, infinite error bounds, overflows). We mark in bold the smaller number of fails per comparison, and median values where DS2L did better than competitors. Note that we provide comparison on small and medium benchmarks for completeness, while our focus lays on large benchmarks.

In addition to normalized values, we present absolute values of our experiments on large benchmarks in Table 3. 'TO' denotes timeouts, other times are reported in seconds. We additionally mark the benchmarks, for which a tool reported overflow or an infinite error bound. For the original Satire benchmark *lorentz*, the missing configurations *Diff10P, Diff30P, AllDiff* with individual ranges for input DS elements are marked with 'na' (non-applicable). Another original benchmark *heat1d* is only defined for a small size of input DS. We provide absolute experimental values for small and medium benchmarks in the appendix.

Accuracy As expected, state-of-the-art tools often computed tighter error bounds on small and medium benchmarks. However, DS2L was consistently more accurate on the *stdDeviation* benchmark, and the larger (among the two in our set) neural network *controllerTora*. Additionally, Fluctuat reports infinite errors on all medium and large-sized variations of the FFT filter (*fftvector*, *fftmatrix*), while DS2L successfully computes rounding error bounds. Both Fluctuat and DS2L implement—in principle—the same analysis *on the unrolled programs*, and the DS abstractions do not affect accuracy (see Section 6.3). The differences

Table 3: Experimental results on large benchmarks. Reported error bounds are rounded to two digits after decimal point, time is in seconds. "TO" denotes a timeout, "na" stands for non-applicable. **Bold** marks 'winning' values.

	AllSame		*Diff10P*		*Diff30P*		*AllDiff*	
Benchmark	error	time	error	time	error	time	error	time
DS2L								
avg	5.82e-11	**1.90**	2.86e-11	**5.48**	1.87e-11	**19.73**	**1.51e-11**	**157.63**
variance	7.39e-05	**119.28**	1.92e-05	**248.64**	9.68e-06	**412.39**	6.37e-06	**1144.74**
stdDev.	9.01e+03	**118.98**	9.35e-06	**244.51**	4.86e-07	**410.50**	2.70e-07	**1141.30**
roux1	**7.21e-14**	**3.86**	2.46e-13	**10.78**	2.64e-13	**29.88**	2.32e-13	**184.44**
goubault	7.46e-14	**3.93**	8.39e-14	**9.92**	8.39e-14	**27.98**	8.39e-14	**172.90**
harmonic	3.64e-08	**6.55**	1.42e-08	**28.40**	1.13e-08	**100.32**	1.12e-08	**713.85**
nonlin1	*overflow*	-	*overflow*	-	*overflow*	-	*overflow*	-
nonlin2	*overflow*	-	*overflow*	-	*overflow*	-	*overflow*	-
nonlin3	1.08e+74	**766.73**	2.66e+73	**1446.47**	-	TO	-	TO
pendulum	4.69e+81	**1583.05**	-	TO	-	TO	-	TO
heat1d	1.14e-13	**222.82**	7.26e-14	**830.00**	7.14e-14	**874.28**	7.13e-14	**857.08**
conv.2d_size3	3.15e-10	**63.29**	2.88e-10	**111.15**	2.62e-10	**158.94**	-	TO
sobel3	*DivByZero*	-	*DivByZero*	-	*DivByZero*	-	*DivByZero*	-
fftmatrix	**4.39e-08**	325.93	**4.24e-08**	387.76	**3.95e-08**	386.29	**2.97e-08**	399.04
fftvector	**2.02e-08**	262.99	**1.62e-08**	266.15	**1.13e-08**	269.89	**9.84e-09**	278.46
lorentz	3.42e-12	2.33	3.27e-12	2.32	3.27e-12	2.21	1.25e-12	2.27
alphaBlend.	3.14e-13	**1.83**	3.14e-13	**47.81**	3.14e-13	**225.50**	-	TO
contr.Tora	2.61e-04	**386.38**	-	TO	-	TO	-	TO
lyapunov	7.02e-08	**104.21**	-	TO	-	TO	-	TO
Fluctuat								
avg	**2.57e-11**	516.00	**1.83e-11**	475.50	**1.63e-11**	462.00	**1.51e-11**	490.50
variance	-	TO	-	TO	-	TO	-	TO
stdDev.	-	TO	-	TO	-	TO	-	TO
roux1	2.10e-13	1310.00	**2.10e-13**	1302.00	**7.52e-14**	1212.50	**1.09e-13**	1295.50
goubault	**6.50e-14**	695.50	**6.45e-14**	726.50	**6.45e-14**	711.00	**1.87e-14**	716.50
harmonic	-	TO	-	TO	-	TO	-	TO
nonlin1	-	TO	-	TO	-	TO	-	TO
nonlin2	-	TO	-	TO	-	TO	-	TO
nonlin3	-	TO	-	TO	-	TO	-	TO
pendulum	-	TO	-	TO	-	TO	-	TO
heat1d	-	TO	-	TO	-	TO	-	TO
conv.2d_size3	-	TO	-	TO	-	TO	-	TO
sobel3	-	TO	-	TO	-	TO	-	TO
fftmatrix	∞	71.33	∞	71.33	∞	71.67	∞	71.00
fftvector	∞	35.67	∞	37.67	∞	116.00	∞	107.33
lorentz	**1.23e-13**	**2.00**	**1.21e-13**	**2.00**	**1.21e-13**	**2.00**	**1.02e-13**	**1.50**
alphaBlend.	-	TO	-	TO	-	TO	-	TO
contr.Tora	-	TO	-	TO	-	TO	-	TO
lyapunov	-	TO	-	TO	-	TO	-	TO
Satire								
avg	3.47e-11	1456.27	2.72e-11	1421.63	2.22e-11	1419.89	2.02e-11	1410.32
variance	-	TO	-	TO	-	TO	-	TO
stdDev.	-	TO	-	TO	-	TO	-	TO
roux1	-	TO	-	TO	-	TO	-	TO
goubault	-	TO	-	TO	-	TO	-	TO
harmonic	-	TO	-	TO	-	TO	-	TO
nonlin1	-	TO	-	TO	-	TO	-	TO
nonlin2	-	TO	-	TO	-	TO	-	TO
nonlin3	-	TO	-	TO	-	TO	-	TO
lorentz	1.35e-13	1327.65	*na*	*na*	*na*	*na*	*na*	*na*

in accuracy come from 1) the optimized evaluation of `fold`s; 2) DS2L's use of intervals instead of affine arithmetic; and 3) internal implementation differences that for the closed-source Fluctuat are not evident. We note that both Fluctuat's and DS2L's reported errors are itself small, and thus practically useful.

Satire reported more accurate results for non-linear benchmarks. On two configurations where DS2L reported overflow for the small input DS size (*AllSame, Diff10P* for *nonlin1* and *Diff10P, Diff30P* for *nonlin2*), Satire successfully reported rounding errors. Predictably, on benchmarks where DS2L used over-approximation of `fold`s Satire's reported errors were also smaller. However, on all linear benchmarks except *harmonic* DS2L's accuracy could be recovered by using a non-optimized evaluation of `fold` (while still being faster than Satire, but by a smaller factor). Despite the over-approximation, DS2L was consistently more accurate on the linear *goubault*. Interestingly, DS2L was 3x more accurate than Satire on its original benchmark *heat1d*. Note that the *original* benchmark *heat1d* corresponds to the *AllSame* specification granularity and the small DS size. Experimental data for this setting is available in the appendix in Table 4.

Performance The performance comparison shows that DS2L scales better to larger programs: it reports results on 50% more *large* benchmarks than Fluctuat and on 59% more than Satire. Additionally, DS2L is faster than Fluctuat on most large and medium-sized benchmarks with a median speedup factor of 25x and 2x respectively. A notable outlier is *alphaBlending*, where DS2L is **4636x** faster than Fluctuat. This is due to the benchmark's internal structure: it contains element-wise operations on matrices, where DS2L's abstraction is particularly efficient.

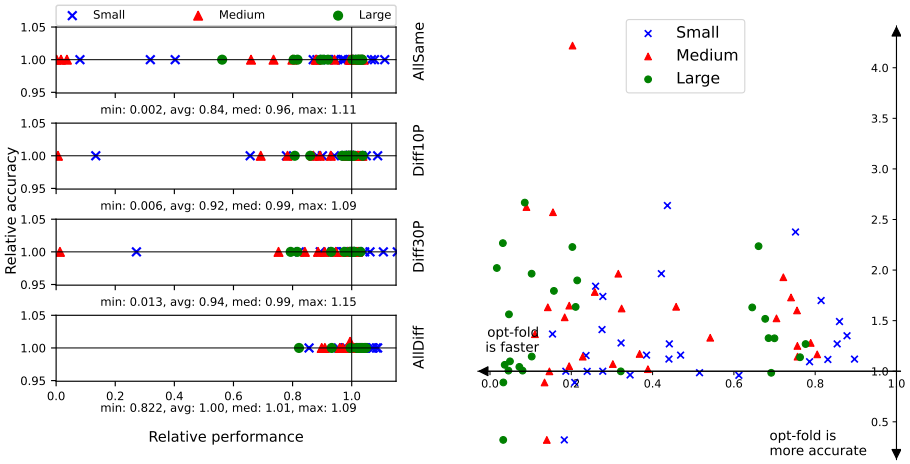
Satire timed out more often than DS2L on all sizes of benchmarks, and particularly on large benchmarks where it failed to report results on all benchmarks except *avg* and *lorentz* (see Table 3). Moreover, Satire was slower than DS2L by at least **6x** and median **36x** across different sizes of benchmarks including its original benchmarks *heat1d* and *lorentz*.

RQ1 Conclusion: Based on our experimental data, we conclude that DS2L is *significantly faster* than Satire and specifically *scales better* than Fluctuat and Satire to larger programs and is consequently able to report an error for more and larger benchmarks. While DS2L is often less accurate than Fluctuat and Satire, it still produces meaningful accuracy bounds.

6.3 RQ2: DS-based Abstraction Accuracy/Performance Tradeoff

Our analysis differs from the analysis of the unrolled programs in two main points: it leverages the DS abstraction, and optimizes the evaluation of `fold`s (Section 4.4). We evaluate the effect of these differences on both accuracy and performance. We split this evaluation into two parts: first, we check the effect of the DS abstraction alone, then we examine the benefits of the optimized `fold`s.

DS Abstraction. First, we compare the DS abstraction-based analysis of DS2L to a baseline analysis that works on unrolled code. To avoid confounding factors such as programming language choice, analysis type etc., we do this



(a) DS2L with abstraction vs. the baseline analysis

(b) DS2L: Optimized fold vs. unrolled

Fig. 4: Relative performance/accuracy of DS2L in various configurations

comparison on a baseline analysis that we implement within DS2L itself and that shares exactly its analysis for straight-line code. We denote this baseline analysis by BASE. BASE internally unrolls all operations, and thus just like DS2L does not explicitly construct an AST for the entire program, as this may be unnecessarily costly and bias the results. Thus, when comparing DS2L and BASE, the only difference consists in using the corresponding DS abstractions during the analysis. For the purpose of DS abstraction evaluation we use the version of DS2L without over-approximation on `fold`s.

Specifically, we compare normalized analysis time and normalized computed worst-case absolute rounding errors per benchmark for each of its 12 variants. Figure 4a summarizes the results, smaller values on both axes are better. The x-axis shows relative analysis time of the DS abstraction analysis to the baseline, values with $x < 1$ denote benchmarks, on which DS2L was faster than BASE. The y-axis represents relative accuracy, values with $y = 1$ show that the worst-case rounding errors reported by DS2L were exactly the same as for BASE. We provide average, median, minimum and maximum relative analysis times for each specification.

For most benchmarks applying the DS abstraction has improved the analysis performance. Predictably, the performance boost was stronger for coarser specifications and close to none on the *AllDiff* specification that assigns each DS element an individual input range. We manually checked the cases where DS2L was slower than BASE. For these cases the absolute time difference is under 0.3 seconds on small and medium configurations (up to 15% of analysis time), and under 72 seconds on large configurations (at most 5% of the analysis time). We attribute this to the normal variation in running times and do not see it as a systematic problem.

The computed errors were the same for DS2L and BASE on all benchmarks. This result confirms our expectation that the DS abstractions (without `fold` optimizations) do not change the semantics and therefore do not affect computed rounding errors.

Optimized folds. We evaluate the effect of our `fold` optimization on top of DS abstraction improvements in Figure 4b. We compare the relative accuracy and performance on benchmarks with `fold` with and without the optimization. As expected, the optimized `fold` evaluation is faster and less accurate on most benchmarks, these are the points above the x-axis and to the left of the y-axis. The effect is more pronounced on the large benchmarks. Interestingly, in some cases the optimized evaluation reported smaller error bounds despite introducing an over-approximation of ranges. Upon closer inspection we note that some of the randomly generated input range bounds cannot be exactly represented in floating points, hence performing an unrolled error computation on such ranges will include the bounds' rounding error and magnify it (artificially) in subsequent iterations. The accuracy can thus improve in cases where the over-approximated ranges were exactly representable in floats, while corresponding element's input ranges were not.

RQ2 Conclusion: The DS abstraction alone improves the analysis' performance while having no effect on the accuracy. A user may further improve the performance by providing a coarser specification or enabling the optimized evaluation of `fold`s, which trades off accuracy for performance.

7 Related Work

Besides Fluctuat [13] and Satire [9], several other tools exist for computing guaranteed upper bounds on rounding errors; Gappa [10], Daisy [7], FPTaylor [30], Real2Float [24], Rosa [8] and PRECiSA [31]. These either implement a dataflow analysis based approach very similar to Fluctuat's or an optimization-based approach similar to Satire. Most of the research has focused on analyzing straight-line numerical expressions as accurately as possible, i.e. computing error bounds as close to the actual errors as possible. Of these, Satire has been shown to be most scalable [9].

A few of these tools can also handle limited programs beyond straight-line expressions. As already discussed, Fluctuat [13] can handle loops via unrolling or with widening, but as we observed widening has limited success with a complex analysis such as the one used to analyze floating-point rounding errors. Rosa [8] provides a more efficient way to bound rounding errors in bounded loops than complete unrolling for a specific type of while loops, but requires invariants about the variable's ranges to be given. Rosa [8] and PRECiSA [31] also support (simple) conditional branches where they also compute the error due to diverging executions between then- and else-branches, in addition to rounding errors of each individual branch. Such techniques are complementary to DS2L's handling of data structures.

In contrast to sound analysis tools, dynamic analysis tools for floating-point programs have less restrictions on the input programs and generally handle whole programs, including loops, conditional branches and data structures. Typically, they execute a program on particular floating-point inputs side-by-side with a shadow execution in a higher precision [2,4,32], for instance implemented using arbitrary-precision arithmetic, that serves as an approximation of the ideal real-valued execution. By their nature, dynamic analyses cannot compute guaranteed bounds on errors, only an estimate of the errors for inputs tried. Several tools use a dynamic analysis to identify inputs that result in particularly large rounding errors [4,32]. Symbolic execution has also been used to find inputs that cause overflow or large precision loss in floating-point programs [21,14,1]. Recent work also combines dynamic and static analysis for identifying, or showing conditional absence of large rounding errors in larger floating-point programs [22].

Abstract interpretation based analyzers such as the industrial-strength Astrée [26], or implementations of different numerical domains such as Apron [19] and ELINA [28] can prove safety of floating-point programs, i.e. the absence of overflows, division-by-zero or out-of-bounds errors by bounding the ranges of variables. They do not, however, quantify rounding errors.

8 Conclusion

We have shown that computing rounding errors over a functional representation of floating-point list programs can be beneficial for analysis performance, by leveraging implicit semantic information present in the high-level representation. Conceptually, our idea appears simple - "just" use a functional input language - and yet, it has not been pursued before. We view this simplicity as a strength, but also note that a effective realization of this idea required a careful design of the DSL and the analysis, as well as substantial implementation effort. Our analysis can generally handle more, and especially larger benchmarks, though some of this performance benefit comes at a trade-off with analysis accuracy. Future work should determine whether it is possible to recover some of this accuracy with minimum performance loss.

A Appendix

Domain Specific Language Syntax We provide a representative subset of our domain-specific language in Figure 5. The syntax of all binary arithmetic operations is the same, we therefore omit repeating occurrences.

Experimental Data We provide the experimental results used to evaluate DS2L in Section 6. Table 4 shows results for the small input DSs, Table 5 for medium. Whenever a tool has failed to report the error bound we use "-" to denote it, we also indicate reported *overflow* explicitly, we write ∞ if the reported error bounds were $[-\infty, \infty]$. *'DivByZero'* denotes the case when the analysis detected that the denominator range may include zero. We use "TO" to denote 30-minute timeouts and any other tool failures. The reported time is

```
object Vector {
  def zeroVector(i:Int): Vector
  def zip(v1: Vector, v2: Vector): Matrix
}
case class Vector(data: List[Real]) {
  // uncertainty on the vector
  def +/-(x: Real): Boolean
  // specify one range for the whole vector
  def <=(x: Real): Boolean // also >=
  // specify range for a subset of elements
  def specV(ranges: Set[((Int, Int), (Real, Real))]):
    Boolean
  def size(i: Int): Boolean
  // element-wise operations
  def +(v: Vector): Vector // also -,*,/
  // element-wise elementary functions
  def log(): Vector // sin(), cos(), tan(), ctan(), etc.
  // cross-product
  def x(v: Vector): Vector
  // operations with constants
  def *(c: Real): Vector // also +, /
  // non-arithmetic operations
  def length(): Int
  def at(i: Int): Real
  def slice(i: Int, j: Int): Vector
  def everyNth(i: Int, from: Int): Vector
  // standard functions
  def map(fnc: (Real) => Real): Vector
  def fold(init: Real)(fnc: (Real,Real) => Real): Real
  def filter(fnc: (Real) => Boolean): Vector
  // sliding window
  def slideReduce(size:Int, step: Int)(
    fnc:(Vector) => Real): Vector
  def enumSlideFlatMap(n: Int)(
    fnc: (Int, Vector) => Vector): Vector
  // add zeros padding around the vector
  def pad(i: Int): Vector
  def max(): Real // also min()
  def sum(): Real // syntactic sugar for fold(0.0)(λa,x.a+x)
  // concatenate and add elements
  def ++(v: Vector): Vector // also append :+(_), prepend +:(_)
}
object Matrix {
  def zeroMatrix(i:Int, j:Int): Matrix
}
case class Matrix(data: List[List[Real]]) {
  // < same as in Vector >
  // element-wise operations and elem. functions
  // operations with constants
  // non-arithmetic operations
  // basic functional ops
  // < different from Vector >
  // input spec for range and size
  def specM(ranges: Set[(Set[(Int, Int)], (Real, Real))]):
    Boolean
  def size(i: Int, j: Int): Boolean
  // +non-arithmetic operations
  def row(i: Int): Vector
  def slice(fromI: Int, fromJ: Int)(toI: Int, toJ: Int): Matrix
  def at(i:Int, j: Int): Real
  def numRows(): Int
  def numCols(): Int
  // flip elements upside down
  def flipud(): Matrix // also fliplr() left to right
  def enumRowsMap(fnc: (Int, Vector) => Vector): Matrix
  // operations on individual elements
  def mapElements(fnc: (Real) => Real): Matrix
  def foldElements(init: Real)(fnc: (Real,Real) => Real): Real
}
```

Fig. 5: DSL for numerical programs on data structures

the analysis time in seconds. "*na*" in Satire's results denotes that we did not run Satire on these variations of *heat1d* or *lorentz*, as we only took the original benchmarks that had the same ranges for all input DS elements.

References

1. Barr, E.T., Vo, T., Le, V., Su, Z.: Automatic detection of floating-point exceptions. In: Principles of Programming Languages (POPL) (2013)
2. Benz, F., Hildebrandt, A., Hack, S.: A Dynamic Program Analysis to Find Floating-Point Accuracy Problems. In: Programming Language Design and Implementation (PLDI) (2012)
3. Cheng, T., Rival, X.: An Abstract Domain to Infer Types over Zones in Spreadsheets. In: Static Analysis Symposium (SAS) (2012)
4. Chiang, W., 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) (2014)
5. Cousot, P., Cousot, R., Logozzo, F.: A parametric segmentation functor for fully automatic and scalable array content analysis. In: Principles of Programming Languages (POPL) (2011)
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: Numerical Software Verification Workshop (NSV) (2016)
7. Darulova, E., Izycheva, A., Nasir, F., Ritter, F., Becker, H., Bastian, R.: Daisy - Framework for Analysis and Optimization of Numerical Programs. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (2018)
8. Darulova, E., Kuncak, V.: Towards a compiler for reals. ACM Trans. Program. Lang. Syst. **39**(2) (2017)
9. Das, A., Briggs, I., Gopalakrishnan, G., Krishnamoorthy, S., Panchekha, P.: Scalable yet rigorous floating-point error analysis. In: International Conference for High Performance Computing, Networking, Storage and Analysis (SC) (2020)
10. De, D., Lauter, C., Melquiond, G.: Assisted Verification of Elementary Functions Using Gappa. In: ACM Symposium on Applied Computing (2006)
11. de Figueiredo, L.H., Stolfi, J.: Affine arithmetic: Concepts and applications. Numer. Algorithms **37**(1-4) (2004)
12. Fousse, L., Hanrot, G., Lefèvre, V., Pélissier, P., Zimmermann, P.: Mpfr: A multiple-precision binary floating-point library with correct rounding. ACM Trans. Math. Softw. **33**(2) (2007)
13. Goubault, E., Putot, S.: Static Analysis of Finite Precision Computations. In: Verification, Model Checking, and Abstract Interpretation (VMCAI) (2011)
14. Guo, H., Rubio-González, C.: Efficient Generation of Error-Inducing Floating-Point Inputs via Symbolic Execution. In: International Conference on Supercomputing (ICS) (2020)
15. Hagedorn, B., Stoltzfus, L., Steuwer, M., Gorlatch, S., Dubach, C.: High performance stencil code generation with lift. In: Proceedings of the 2018 International Symposium on Code Generation and Optimization (2018)
16. Halbwachs, N., Péron, M.: Discovering properties about arrays in simple programs. In: Programming Language Design and Implementation (PLDI) (2008)
17. IEEE, C.: IEEE Standard for Floating-Point Arithmetic. IEEE Std 754-2008 (2008)

Table 4: Experimental results on small benchmarks. Reported error bounds are rounded to two digits after decimal point, time is in seconds. "TO" denotes a timeout, "na" stands for non-applicable.

	AllSame		*Diff10P*		*Diff30P*		*AllDiff*	
Benchmark	error	time	error	time	error	time	error	time
DS2L								
avg	4.62e-13	0.14	2.60e-13	0.17	1.59e-13	0.20	1.42e-13	0.26
variance	5.98e-07	1.06	2.62e-07	1.16	8.82e-08	1.17	6.17e-08	1.32
stdDev.	145	0.92	2.81e-08	1.16	3.00e-09	1.09	3.96e-09	1.23
roux1	7.21e-14	0.22	2.26e-13	0.29	2.27e-13	0.34	2.07e-13	0.55
goubault	7.46e-14	0.22	8.38e-14	0.29	7.97e-14	0.33	8.34e-14	0.57
harmonic	1.15e-11	0.31	5.84e-12	0.39	5.81e-12	0.59	5.82e-12	1.09
nonlin1	*overflow*	-	*overflow*	-	1.17e-07	4.07	1.06e-09	4.43
nonlin2	*overflow*	-	*overflow*	-	*overflow*	-	*overflow*	-
nonlin3	7.82e-14	3.85	3.72e-14	4.13	3.18e-14	4.12	2.78e-14	4.55
pendulum	2.27e-13	2.68	2.27e-13	2.74	7.51e-14	3.26	1.73e-13	3.50
heat1d	7.33e-15	1.65	4.79e-15	2.24	4.64e-15	2.22	4.94e-15	2.48
conv.2d_size3	3.15e-10	0.28	1.51e-10	0.30	1.51e-10	0.33	4.07e-11	0.50
sobel3	*DivByZero*	-	*DivByZero*	-	*DivByZero*	-	*DivByZero*	-
fftmatrix	4.29e-12	1.00	4.29e-12	0.95	4.29e-12	1.08	1.88e-12	1.06
fftvector	2.85e-12	0.96	2.40e-12	0.83	1.32e-12	0.89	1.86e-12	0.86
lorentz	4.33e-14	1.29	4.33e-14	1.19	2.55e-14	1.30	3.77e-14	1.28
alphaBlend.	3.14e-13	0.07	3.14e-13	0.09	3.14e-13	0.17	3.13e-13	0.74
contr.Tora	1.42e-12	0.43	1.42e-12	1.01	1.20e-12	1.52	1.74e-13	1.38
lyapunov	9.90e-13	0.28	8.87e-13	0.49	6.64e-13	0.67	1.68e-13	0.59
Fluctuat								
avg	2.62e-13	0.11	1.99e-13	0.07	1.62e-13	0.07	1.41e-13	0.07
variance	1.15e-09	6.33	6.17e-10	6.00	3.25e-10	6.00	2.16e-10	6.00
stdDev.	516	6.67	1.35e-11	6.00	5.38e-12	6.00	7.13e-12	6.00
roux1	2.10e-13	0.16	1.83e-13	0.12	1.46e-13	0.12	6.80e-14	0.12
goubault	6.50e-14	0.13	6.50e-14	0.09	2.43e-14	0.09	4.67e-14	0.09
harmonic	2.92e-12	0.34	2.47e-12	0.21	2.25e-12	0.21	2.11e-12	0.21
nonlin1	3.07e-14	16.33	2.78e-14	13.00	2.48e-14	13.00	2.73e-14	13.67
nonlin2	∞	16.67	2.83e-12	16.33	1.86e-13	16.67	∞	14.33
nonlin3	7.84e-15	0.55	4.29e-15	0.36	3.53e-15	0.38	3.44e-15	0.48
pendulum	2.15e-13	0.36	2.15e-13	0.20	6.67e-14	0.23	1.63e-13	0.24
heat1d	6.66e-16	12.67	4.44e-16	12.33	4.44e-16	12.67	4.44e-16	12.67
conv.2d_size3	1.24e-10	0.08	6.23e-11	0.08	6.23e-11	0.08	1.56e-11	0.08
sobel3	91.9	0.24	61.6	0.24	61.6	0.24	31.0	0.24
fftmatrix	1.09e-12	0.01	1.09e-12	0.01	1.09e-12	0.01	6.92e-13	0.01
fftvector	6.93e-13	0.01	6.93e-13	0.01	3.46e-13	0.01	6.90e-13	0.00
lorentz	2.16e-14	0.23	2.16e-14	0.23	1.55e-14	0.22	1.90e-14	0.23
alphaBlend.	1.56e-13	0.11	1.56e-13	0.12	1.56e-13	0.12	1.55e-13	0.12
contr.Tora	2.28e-12	0.57	1.80e-12	0.58	1.25e-12	0.56	1.17e-13	0.49
lyapunov	8.07e-13	0.09	7.08e-13	0.09	4.58e-13	0.10	9.34e-14	0.10
Satire								
avg	3.65e-13	1.74	2.93e-13	2.22	2.47e-13	2.07	1.93e-13	1.83
variance	1.92e-09	12.22	1.19e-09	16.23	7.57e-10	16.04	4.97e-10	16.54
stdDev.	-	TO	-	TO	-	TO	-	TO
roux1	2.55e-13	11.61	2.26e-13	13.39	1.88e-13	13.27	8.58e-14	13.18
goubault	1.14e-13	13.44	1.14e-13	15.37	4.97e-14	14.77	9.74e-14	14.55
harmonic	7.90e-12	19.48	5.01e-12	21.31	4.66e-12	20.48	4.90e-12	20.74
nonlin1	4.28e-14	655.23	3.99e-14	678.74	3.49e-14	687.85	3.87e-14	700.75
nonlin2	-	TO	1.94e-15	100.92	1.94e-15	89.29	-	TO
nonlin3	8.43e-15	32.45	5.62e-15	27.50	4.90e-15	26.09	3.86e-15	28.80
lorentz	9.68e-15	577.99	*na*	*na*	*na*	*na*	*na*	*na*
heat1d	1.98e-14	92.76	*na*	*na*	*na*	*na*	*na*	*na*

Table 5: Experimental results on medium benchmarks. Reported error bounds are rounded to two digits after decimal point, time is in seconds. "TO" denotes a timeout, "na" stands for non-applicable.

	AllSame		*Diff10P*		*Diff30P*		*AllDiff*	
Benchmark	error	time	error	time	error	time	error	time
DS2L								
avg	3.65e-12	0.47	3.41e-12	0.60	2.01e-12	0.84	1.52e-12	2.25
variance	4.64e-06	4.95	2.42e-06	7.05	7.82e-07	8.66	6.16e-07	14.04
stdDev.	1.13e+03	4.96	2.49e-07	6.96	3.11e-08	8.47	3.00e-08	14.27
roux1	7.21e-14	0.84	2.29e-13	1.21	2.41e-13	1.80	2.30e-13	4.04
goubault	7.46e-14	0.83	8.38e-14	1.09	8.39e-14	1.69	8.36e-14	3.77
harmonic	4.56e-10	1.29	1.94e-10	2.03	1.71e-10	3.21	1.59e-10	9.66
nonlin1	*overflow*	-	*overflow*	-	*overflow*	-	*overflow*	-
nonlin2	*overflow*	-	*overflow*	-	*overflow*	-	*overflow*	-
nonlin3	4.32e-05	32.17	2.91e-05	40.75	1.63e-05	49.87	1.16e-05	76.33
pendulum	1.16e-04	28.02	5.50e-05	37.76	5.50e-05	48.87	6.65e-05	85.65
heat1d	1.44e-14	4.20	9.23e-15	6.61	9.09e-15	6.81	9.10e-15	6.82
conv.2d_size3	3.15e-10	0.90	1.68e-10	1.11	2.62e-10	1.43	6.76e-11	2.27
sobel3	*DivByZero*	-	*DivByZero*	-	*DivByZero*	-	*DivByZero*	-
fftmatrix	3.71e-09	55.34	3.66e-09	57.13	3.58e-09	57.89	2.56e-09	59.44
fftvector	1.84e-09	39.84	1.48e-09	39.56	1.12e-09	39.56	1.05e-09	41.12
lorentz	2.28e-13	1.73	2.17e-13	1.69	1.53e-13	1.79	1.84e-13	1.75
alphaBlend.	3.14e-13	0.19	3.14e-13	1.00	3.14e-13	1.95	3.14e-13	773.52
contr.Tora	8.75e-08	6.65	6.14e-08	468.52	4.06e-08	485.40	7.33e-09	962.96
lyapunov	5.88e-10	2.90	4.16e-10	96.85	3.37e-10	97.68	1.04e-10	198.28
Fluctuat								
avg	2.37e-12	3.75	2.06e-12	4.00	1.75e-12	3.00	1.49e-12	3.50
variance	-	TO	-	TO	-	TO	-	TO
stdDev.	-	TO	-	TO	-	TO	-	TO
roux1	2.10e-13	8.67	8.16e-14	8.00	5.15e-14	8.00	9.86e-14	8.00
goubault	6.50e-14	5.00	6.50e-14	4.67	6.09e-14	5.00	5.91e-14	5.00
harmonic	1.68e-10	16.00	1.34e-10	14.67	1.16e-10	15.00	1.09e-10	16.00
nonlin1	-	TO	-	TO	-	TO	-	TO
nonlin2	-	TO	-	TO	-	TO	-	TO
nonlin3	2.21e-08	29.67	8.24e-09	30.00	6.02e-09	32.33	6.92e-09	35.50
pendulum	1.12e-04	13.00	5.22e-05	13.67	5.22e-05	16.33	6.33e-05	20.00
heat1d	6.66e-16	236.33	4.44e-16	249.33	4.44e-16	248.33	4.44e-16	247.33
conv.2d_size3	1.24e-10	1.00	1.07e-10	1.33	1.24e-10	1.00	4.09e-11	1.33
sobel3	91.9	6.67	90.3	6.33	86.3	6.67	54.4	6.33
fftmatrix	∞	2.67	∞	2.67	∞	2.67	∞	2.67
fftvector	∞	0.99	∞	2.67	∞	2.67	∞	2.33
lorentz	5.41e-14	0.73	5.28e-14	0.73	4.73e-14	0.73	4.84e-14	0.83
alphaBlend.	1.56e-13	744.67	1.56e-13	843.67	1.56e-13	820.33	1.55e-13	743.67
contr.Tora	-	TO	-	TO	-	TO	-	TO
lyapunov	5.69e-10	264.00	4.01e-10	270.50	3.03e-10	269.00	7.89e-11	258.00
Satire								
avg	3.49e-12	18.70	2.84e-12	15.66	2.34e-12	14.45	2.05e-12	15.99
variance	1.83e-08	476.62	1.09e-08	457.62	6.11e-09	455.67	4.61e-09	458.18
stdDev.	-	TO	-	TO	-	TO	-	TO
roux1	2.55e-13	126.46	1.19e-13	118.79	7.00e-14	122.02	1.64e-13	120.40
goubault	1.14e-13	90.19	1.14e-13	83.96	1.10e-13	82.39	9.79e-14	82.82
harmonic	1.62e-11	93.69	1.29e-11	90.72	1.07e-11	89.45	1.03e-11	90.97
nonlin1	-	TO	-	TO	-	TO	-	TO
nonlin2	-	TO	-	TO	-	TO	-	TO
nonlin3	9.17e-15	491.27	6.04e-15	496.15	4.13e-15	468.28	4.94e-15	482.47
lorentz	3.50e-14	875.69	*na*	*na*	*na*	*na*	*na*	*na*
heat1d	*na*	92.76	*na*	*na*	*na*	*na*	*na*	*na*

18. Izycheva, A., Darulova, E.: On sound relative error bounds for floating-point arithmetic. In: Formal Methods in Computer Aided Design (FMCAD) (2017)
19. Jeannet, B., Miné, A.: Apron: A library of numerical abstract domains for static analysis. In: Computer Aided Verification (CAV) (2009)
20. Johnson, T.T., Lopez, D.M., Musau, P., Tran, H., Botoeva, E., Leofante, F., Maleki, A., Sidrane, C., Fan, J., Huang, C.: Arch-comp20 category report: Artificial intelligence and neural network control systems (ainncs) for continuous and hybrid systems plants. In: International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH20) (2020)
21. Liew, D., Schemmel, D., Cadar, C., Donaldson, A., Zähl, R., Wehrle, K.: Floating-Point Symbolic Execution: A Case Study in N-Version Programming. In: ASE (2017)
22. Lohar, D., Jeangoudoux, C., Sobel, J., Darulova, E., Christakis, M.: A two-phase approach for conditional floating-point verification. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (2021)
23. M.Abadi, A.Agarwal, P.Barham, E.Brevdo, Z.Chen, C.Citro, G.S.Corrado, A.Davis, J.Dean, M.Devin, S.Ghemawat, I.Goodfellow, A.Harp, G.Irving, M.Isard, Y.Jia, R.Jozefowicz, L.Kaiser, M.Kudlur, J.Levenberg, D.Mané, R.Monga, S.Moore, D.Murray, C.Olah, M.Schuster, J.Shlens, B.Steiner, I.Sutskever, K.Talwar, P.Tucker, V.Vanhoucke, V.Vasudevan, F.Viégas, O.Vinyals, P.Warden, M.Wattenberg, M.Wicke, Y.Yu, X.Zheng: TensorFlow: Large-scale machine learning on heterogeneous systems (2015), `https://www.tensorflow.org/`, software available from tensorflow.org
24. Magron, V., Constantinides, G., Donaldson, A.: Certified Roundoff Error Bounds Using Semidefinite Programming. ACM Trans. Math. Softw. **43**(4) (2017)
25. Miné, A., Breck, J., Reps, T.: An algorithm inspired by constraint solvers to infer inductive invariants in numeric programs. In: Programming Languages and Systems (ESOP) (2016)
26. Miné, A., Mauborgne, L., Rival, X., Feret, J., Cousot, P., Kästner, D., Wilhelm, S., Ferdinand, C.: Taking Static Analysis to the Next Level: Proving the Absence of Run-Time Errors and Data Races with Astrée. In: ERTS (2016)
27. Moore, R.E., Kearfott, R.B., Cloud, M.J.: Introduction to Interval Analysis. Society for Industrial and Applied Mathematics (2009)
28. Singh, G., Püschel, M., Vechev, M.T.: Fast polyhedra abstract domain. In: Principles of Programming Languages (POPL) (2017)
29. Solovyev, A., Baranowski, M.S., Briggs, I., Jacobsen, C., Rakamaric, Z., Gopalakrishnan, G.: Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. ACM Trans. Program. Lang. Syst. **41**(1) (2019)
30. Solovyev, A., Jacobsen, C., Rakamaric, Z., Gopalakrishnan, G.: Rigorous Estimation of Floating-Point Round-off Errors with Symbolic Taylor Expansions. In: Formal Methods (FM) (2015)
31. Titolo, L., Feliú, M., Moscato, M., Muñoz, C.: An abstract interpretation framework for the round-off error analysis of floating-point programs. In: Verification, Model Checking, and Abstract Interpretation (VMCAI) (2018)
32. Zou, D., Wang, R., Xiong, Y., Zhang, L., Su, Z., Mei, H.: A Genetic Algorithm for Detecting Significant Floating-Point Inaccuracies. In: International Conference on Supercomputing (ICS) (2015)