Inferring Interval-Valued Floating-Point Preconditions[⋆]

Jonas Krämer[1], Lionel Blatter[2], Eva Darulova[3](✉), and Mattias Ulbrich[2]

[1] itemis AG, Stuttgart, Germany [§], jonas.kraemer@itemis.com
[2] KIT, Karlsruhe, Germany {lionel.blatter,ulbrich}@kit.edu
[3] Uppsala University, Uppsala, Sweden[†], eva.darulova@it.uu.se

Abstract. Aggregated roundoff errors caused by floating-point arithmetic can make numerical code highly unreliable. Verified postconditions for floating-point functions can guarantee the accuracy of their results under specific preconditions on the function inputs, but how to systematically find an adequate precondition for a desired error bound has not been explored so far. We present two novel techniques for automatically synthesizing preconditions for floating-point functions that guarantee that user-provided accuracy requirements are satisfied. Our evaluation on a standard benchmark set shows that our approaches are complementary and able to find accurate preconditions in reasonable time.

1 Introduction

Floating-point arithmetic as defined by the IEEE 754 standard [18] is widely used to approximate real arithmetic in embedded or scientific computing applications. While allowing highly efficient computations, the limited precision of floating-point numbers introduces roundoff errors in every single operation [24]. The aggregated errors in computations where such rounding happens repeatedly are challenging to understand and predict intuitively, so that a variety of techniques and tools [10,14,11,29,21,22] have been developed that bound worst-case roundoff errors. These techniques assume a given floating-point precision, e.g. uniform double precision and a precondition $\psi(\bar{x})$ that bounds a function's possibly multivariate parameters $(\bar{x})$, and automatically compute an upper-bound ε on the function result's absolute roundoff error $(f_{err}(\bar{x}))$[4]:

$$\forall \bar{x}.\ \ \psi(\bar{x}) \rightarrow f_{err}(\bar{x}) \leq \varepsilon \tag{1}$$

Answering the inverse question can be equally useful: given a desired roundoff error bound and precision, for which inputs will the computation's result be

[§] Part of this work was done while the author was at KIT, Germany.
[†] Part of this work was done while the author was at MPI-SWS, Germany.
[⋆] Part of this work was funded by the AESC project supported by the Ministry of Science, Research and Arts Baden-Württemberg (Ref: 33-7533.-9-10/20/1).
[4] We provide more formalization details in the next section.

© The Author(s) 2022
D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13243, pp. 303–321, 2022.
https://doi.org/10.1007/978-3-030-99524-9_16

at least this accurate? That is, given a postcondition specifying the error bound for a floating-point function's result, we want to infer a suitable precondition ψ. Such preconditions can be useful for modular verification of larger floating-point programs, or for efficient implementations: for inputs that satisfy the generated precondition, the function can be evaluated using e.g. efficient double-precision floating-point arithmetic, instead of a more accurate but significantly more expensive arbitrary-precision arithmetic [2] that would have to be used for the remaining input space.

Outside the analysis of floating-point software, the automatic synthesis of preconditions for software components is not a new field of study. Dijkstra's weakest precondition calculus [12], while not originally intended to be used for specification inference, can generate weakest preconditions. However, when applied to a floating-point function, it creates a precondition that still contains the floating-point arithmetic of the analyzed program and is, thus, not simpler than the program itself. Recent approaches (targeting non-floating-point programs) for specification inference [23,28,7,13] similarly do not attempt to abstract from arithmetic operations and their inaccuracies.

This paper introduces two novel techniques for synthesizing *sound and abstract preconditions* for floating-point functions. The inferred preconditions $\psi(\bar{x})$ are sound, by which we mean that they are guaranteed to satisfy Eq. (1) for a user-specified error bound ε. The preconditions are abstract in the sense that they do not contain any floating-point arithmetic operations.

We choose to synthesize *interval-valued* preconditions that bound each function parameter by a lower and an upper bound, i.e. $x \in [a, b]$. Such preconditions avoid floating-point arithmetic, and thus roundoff errors, as evaluating them requires only comparisons with constants. Our preconditions are relatively simple on purpose to ensure compatibility with current sound roundoff verification techniques that internally rely on interval-based abstractions. While more complex, e.g. nonlinear, constraints may be more precise, they are not well-supported by state-of-the-art verifiers and thus their benefit would be (currently) lost.

While we aim to synthesize weak preconditions that cover much of the input space, weakest preconditions are not necessarily helpful in the context of floating-point computations. The reason is that the space of inputs satisfying a postcondition—especially one bounding the roundoff error—is in general highly discontinuous due to the discrete nature of floating-point arithmetic. A weakest precondition would thus consist of a large conjunction, with individual terms often covering only a few values, and would hence not be practically useful. Instead, we aim to find preconditions that balance precision (are as weak as possible) and complexity (are simple and can be evaluated efficiently).

We are not aware of an existing approach for generating such sound floating-point preconditions; we thus choose to introduce and explore two quite different techniques that build on existing dynamic and static floating-point analyses in a novel way. Both approaches start by dynamically *sampling* the analyzed function in order to find likely precondition candidates and then use a verification backend to refine them until their soundness can be guaranteed. The first *recur-*

sive subdivision approach does this by recursively subdividing the input space into increasingly smaller cells, discarding those where sampling shows that the postcondition is not satisfied for the contained inputs, and attempting to verify the rest. Since such generated preconditions may still contain a large number of discontinuous subdomains, we further present an optimization algorithm that soundly approximates the preconditions with significantly simpler expressions that can be evaluated more efficiently. The second *classification tree* approach learns areas of inputs for which the postcondition holds based on a classification tree learned from the dynamic samples, and iteratively *refines* verified preconditions in these areas.

Our approaches guarantee soundness of the generated preconditions by verifying each individual interval domain in the preconditions using a sound floating-point roundoff error analyzer. Our approach is generic in the choice of this tool; we integrate the floating-point verification framework Daisy [10].

We evaluate and compare our proposed approaches on benchmarks from the standard floating-point benchmark suite FPBench [8] and show that the approaches are able to find adequate preconditions that (1) are syntactically simple and cheap to evaluate and (2) are relatively weak, i.e. good approximations of the weakest preconditions covering large areas of the input space, thus balancing complexity and permissiveness. For most benchmarks, our approaches find preconditions in under 20 minutes (and often significantly faster). We demonstrate a possible application of our inferred preconditions for performance improvements on a case study using a kernel from a real-world material sciences code that inspired this work.

Contributions In summary, this paper makes the following contributions:

- Two independent novel inference algorithms that generate interval-valued preconditions for floating-point functions. They are the first of their kind.
- An open-source implementation of both approaches as part of the Daisy floating-point analysis framework.
- An extensive evaluation on 99 benchmarks and a case study showing the effectiveness of our precondition inference.

2 Overview

Before explaining our approaches in detail, we provide a high-level overview using an example. Consider the two-dimensional function `himmilbeau` from the floating-point benchmark suite FPBench [8], introduced to evaluate optimization algorithms [16], and defined as

$$\hat{f}(x_1, x_2) = (x_1^2 + x_2 - 11)^2 + (x_1 + x_2^2 - 7)^2 \ .$$

We denote by $\hat{f} : \mathbb{R}^n \to \mathbb{R}$ the ideal, real-valued specification of the function that a developer may want to compute (where n is the number of function arguments, $n = 2$ for our example). While such a function can in principle be implemented exactly, e.g. using rational arithmetic, such an evaluation is generally slow. Hence, in practice, the function would be implemented in finite precision. In this paper, we consider double-precision (64 bit) IEEE 754 [18] floating-point arithmetic, which is one of the most commonly used finite precisions (though our approach generalizes to other floating-point precisions as well). We denote this finite-precision implementation by $f : \mathbb{F}^n \to \mathbb{F}$.

When evaluating f, each computed intermediate value has to be potentially rounded to a value that is representable in finite precision, introducing a *roundoff error*. While each roundoff error individually is (usually) small, the errors propagate and accumulate during the computation, resulting in potentially large errors on a function's result [20]. It is thus important to be able to make statements about this error, for instance as an absolute error: $f_{err}(\bar{x}) = |\hat{f}(\bar{x}) - f(\bar{x})|, \bar{x} \in \mathbb{F}^n$, where we assume that $\bar{x}$ are 'finite' values and not one of the Not-a-Number or Infinity special floating-point values. Our approach assumes and proves that all computations remain within the number ranges of the chosen floating-point precision and that special values never occur during expression evaluation.

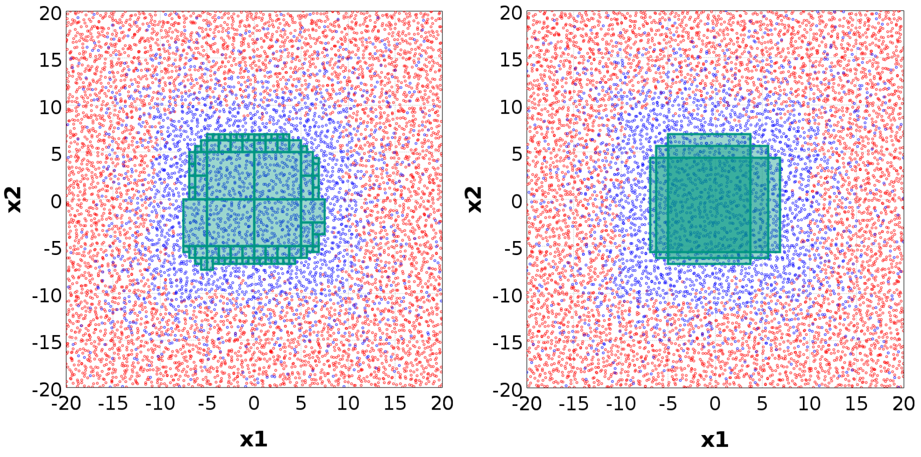
In this paper, we aim to synthesize an interval-valued precondition $\psi(\bar{x})$ that satisfies Eq. (1) ($\forall \bar{x}.\ \psi(\bar{x}) \to f_{err}(\bar{x}) \leq \varepsilon$) where ψ is of the form:

$$\bigvee_{k=1}^{m} \bigwedge_{i=1}^{n} x_i \in [a_{k,i}, b_{k,i}]$$

I.e. such a precondition represents the (set-theoretic) union of m domains of dimension n. To obtain a precondition that can be efficiently checked, we aim to keep m small (< 10), while the precondition should nonetheless be as weak as possible, i.e. cover as much of the input space as possible.

Our precondition inference starts from an initial search area which may be either specified by the user, be defined, for example, by an embedded sensor output domain, or be computed by a static analysis on the call site(s) of f. For our `himmilbeau` example, we assume $x_1, x_2 \in [-20, 20]$ as the search area, and $\varepsilon = 1.4211\text{e-}12$ as the target error bound.

In the first step, our approach samples inputs from the initial search area at random, and evaluates the function f on each input in double precision arithmetic and approximates its corresponding specification $\hat{f}$ using 128 bit arbitrary-precision arithmetic [2]. Comparing the results from the double- and higher-precision evaluations gives us an estimate of the roundoff error. We use this estimate to mark each input as valid or invalid, i.e. as satisfying or violating the postcondition, respectively. Fig. 1 shows the valid and invalid samples for our running example in blue and red, respectively. Note that the error bounds obtained from these samples do not have to be sound, as they are used only for guiding the precondition search; our technique will use static analysis to verify each precondition candidate soundly. Furthermore, the sampling also does not need to identify the exact bounds between valid and invalid samples. As Fig. 1 indicates, such bounds would lead to highly discontinuous preconditions that would be of limited practical use.



(a) Precondition with subdivision (b) Optimized precondition

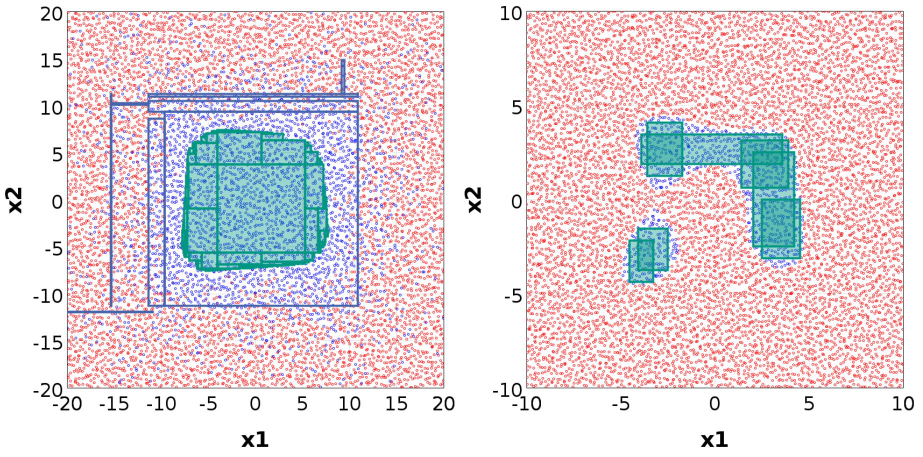
Fig. 1: The sampled `himmilbeau` function. Blue and red points indicate valid and invalid input values, respectively. The rectangles show the inferred preconditions.

Starting from these samples, we explore two techniques. First, we use *interval subdivision* to subdivide the initial search area into equal interval regions (domains such that every dimension is bounded by an interval), and then check each region individually using sound static analysis for whether it is a valid part of the precondition. Fig. 1a shows the generated precondition in green. To reduce the number of regions in the precondition for a simpler and more efficient precondition, we propose an optimization algorithm that approximates the initial verified precondition with fewer, larger regions; the result of this optimization is shown in Fig. 1b.

Subdivision may be inefficient when only a small part of the initial search area constitutes a valid precondition. We thus further explore an approach based on *classification tree learning* that starts from the valid and invalid samples and learns an initial candidate precondition, or a set of candidates if the space of valid samples is disjoint. Then, we again use static error verification to search for sound preconditions. Fig. 2a shows the generated precondition in green.

Ultimately, an inferred precondition allows us to refactor floating-point programs such that they use computations in floats if the result is known to be accurate, and resort to high-precision libraries otherwise. For example, a C-implementation of the `himmilbeau` example using the precondition from Fig. 1b, achieves a 8.6% speed-up against a pure high-precision implementation (on randomly chosen inputs from the range $[-20, 20]$). The precondition that triggers the optimization covers 11.5% of the input domain, hence the size of a precondition nearly directly translates to performance improvements.

The inferred precondition will in general be stronger than the weakest possible precondition, i.e. our inferred preconditions do not cover all of the blue points in Fig. 1 and Fig. 2. There are several reasons: The verification backend



(a) Precondition with classification tree (b) Precondition for range postcondition

Fig. 2: Inferred preconditions for `himmilbeau` using the classification tree approach for the error postcondition, and subdivision for the range postcondition.

has to rely on abstractions and can thus not always verify a valid precondition candidate. Furthermore, due to runtime considerations of our algorithm, the approaches cannot operate on arbitrarily detailed intervals.

Finally, while we discussed our precondition inference for postconditions that target an error bound, our approach equally works for postconditions that specify a target *range*, e.g. that require that the value of the result of our `himmilbeau` function is within given bounds ($f(\bar{x}) \in [-100, 100]$). We show the precondition inferred for this case using subdivision and subsequent optimization in Fig. 2b.

3 Precondition Inference by Subdivision

The first approach that we propose finds preconditions by recursively splitting the initial search area along the parameter axes until it finds interval domains for which the verification backend is able to prove that the target postcondition holds for all inputs. This approach is inspired by interval subdivision that is being used, for example, in roundoff error bound analysis to reduce the amount of over-approximations due to abstractions.

However, a naive application of subdivision for precondition inference is not practical. Each parameter's interval has to be subdivided several times in order to find verifiable preconditions, leading to a large number of regions especially for multi-variate functions. If we then run the relatively expensive verification procedure on each of these regions, the overall running time quickly becomes unreasonable. Furthermore, a precondition consisting of a large number of small interval regions is inefficient to evaluate and unwieldy. We thus combine static and dynamic verification (Sec. 3.1), and optimize the generated preconditions to yield more compact representations (Sec. 3.2).

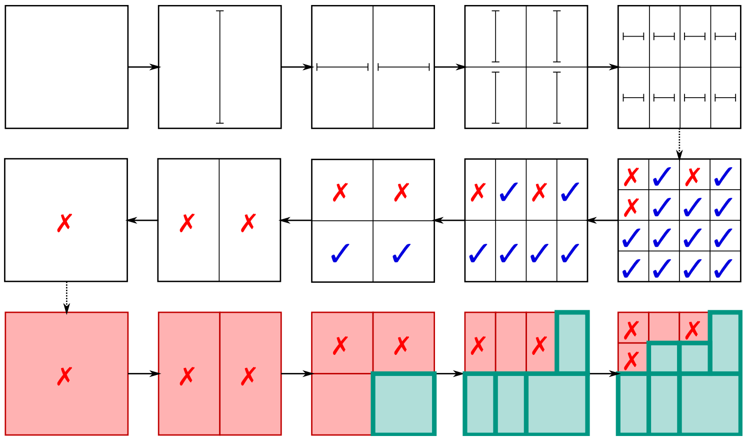


Fig. 3: Illustration of recursive subdivision in two dimensions.

Algorithm 1 Recursive Subdivision

```
1: given arithmetic expression $expr$, postcondition $post$
2: procedure EXTRACTPRE($node$)
3:     if $node \in valid$ then
4:         if verify($node.region, expr, post$) then
5:             return $node.region$
6:     if $node$ is a leaf then return $\emptyset$
7:     else return EXTRACTPRE($n.left$) $\cup$ EXTRACTPRE($n.right$)
```

3.1 Extracting a Verified Precondition from Subdivisions

Our approach starts by building a binary tree, where each node represents an interval region in the search area. The tree is generated by recursively splitting intervals along one parameter axis into two equally sized intervals (called *left* and *right*), splitting along each parameter axis in turn. The top part of Fig. 3 illustrates this subdivision for a two-dimensional example and with a maximum subdivision depth of 4. From left to right, the nodes are repeatedly subdivided until there are 16 leaf nodes.

Our algorithm then runs dynamic sampling (as described in Sec. 2) for each leaf node l. A node l is marked as *valid* (blue check marks in Fig. 3) if the post-condition is satisfied for *all* samples, and as *invalid* (red cross marks) otherwise. The middle part of Fig. 3 shows how these markers ascend to the root of the tree: An inner node i is marked valid if and only if both of its children are valid: $i \in valid \leftrightarrow (i.left \in valid \wedge i.right \in valid)$.

Next, our approach performs a recursive descent (shown in Algorithm 1) from the root node to extract the precondition. The verification backend is queried (`verify` in the algorithm) to verify that intervals are valid (sound) preconditions for all inputs in a given region. As a heurisitic, verification is attempted as close to the root of the tree as possible, as thus a single verification attempt can verify

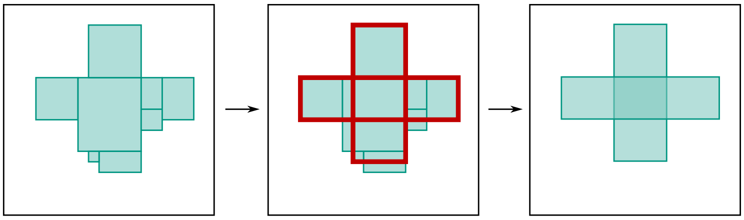


Fig. 4: Approximating generated preconditions.

a larger volume. On the other hand, the verification is more likely to fail, which may increase running time of the algorithm. Verification is futile and thus not attempted for an invalid node ($node \notin valid$). In this case, or if the verification back-end fails to verify, the procedure descends further down the tree.

The bottom part of Fig. 3 illustrates this procedure. No verification is attempted on the root node and its first degree children as they are invalid. Verification is attempted for the two valid grandchild nodes of the root that were marked with a blue check mark. For the lower right node verification is successful, so there is no need to further descend to its child nodes. Verification fails for the left one, which means it has to be subdivided again, like its two remaining sibling nodes. Sometimes subdivision is needed to verify a region even if all of it is ultimately verifiable, such as the lower left region in the last subdivision step. The reason for this is that subdivision generally reduces over-approximations due to the abstractions that the sound verification procedure relies on, and thus often allows to compute tighter error bounds [10].

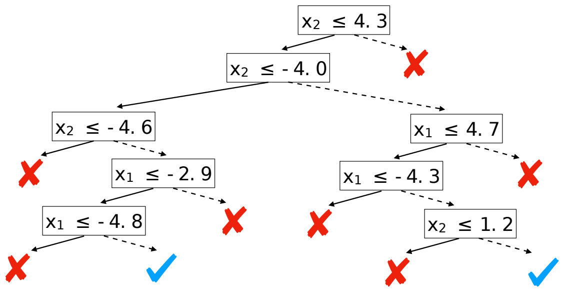
The maximum subdivision depth controls the precision of the approach. With larger depth, the generated preconditions can have a larger volume, i.e. be weaker, but this comes at the cost of a longer running time of the algorithm.

The union of all valid regions extracted from the tree is returned as a precondition. This precondition is sound, since each region has been verified by a sound roundoff error analysis.

3.2 Precondition Optimization

Depending on the subdivision depth, the number of individual regions in a generated precondition can easily reach into the thousands. We observed that one can often approximate the result with significantly fewer regions, while only marginally reducing their volume. Fig. 4 shows an example precondition generated by subdivision on the left, and the optimized precondition on the right. The precondition on the right needs only two regions instead of 8, and covers most of the originally generated precondition and is thus only slightly stronger.

Note that simply picking the largest individual interval regions from the generated precondition is in general insufficient: larger regions may be found within the verified area by composing parts of different intervals into larger ones. While one could in principle use simplification algorithms inside constraint solvers for



(a) Positive/negative decisions have solid/dashed lines. Leaves can be valid (✓) or invalid (✗).

$$(x_1 \in [-4.8, -2.9] \wedge x_2 \in [-4.6, -4.0]) \vee (x_1 \in [-4.2, 4.7] \wedge x_2 \in [1.2, 4.3])$$

(b) Precondition candidate.

Fig. 5: A sample classification tree and the extracted precondition candidate

this task, such algorithms are not targeting our use-case, i.e the smallest formula that covers the biggest valid region.

Thus, we propose an optimization that starts by identifying the interval region that covers the largest verified area and that possibly (partially) covers several interval regions from the originally generated precondition. It then iteratively repeats this process and keeps adding regions that provide the most additional coverage. Since our algorithm is greedy, it is not guaranteed to find an optimal solution, but our experiments have shown that the approximation is very decent even for small numbers of representing regions. Since only regions covering verified areas are added, the optimized precondition is sound. This optimization is also fast compared to the rest of the procedure, since it does not run roundoff verification.

This precondition optimization step can be applied on preconditions obtained from both inferences approaches (recursive subdivision and the refinement approach from the upcoming section), but the effects are more pronounced for the subdivision approach as it usually produces results with more individual regions.

4 Precondition Inference by Decision Tree Learning

Our second precondition inference technique leverages the dynamic samples in a different way: it uses them to generate initial precondition candidates using decision tree learning [4], a well-known algorithm in supervised machine learning. These candidates are subsequently *refined* to obtain sound preconditions. We consider two such refinements in Sec. 4.2 and Sec. 4.3.

4.1 Extracting Candidates from a Classification Tree

First, our algorithm samples the search area as described in Sec. 2, and marks each sample as *valid* or *invalid* depending on whether or not it satisfies the postcondition. The marked or 'classified' samples serve as the training data to train a classification tree (CT) using decision tree learning. A CT is a binary

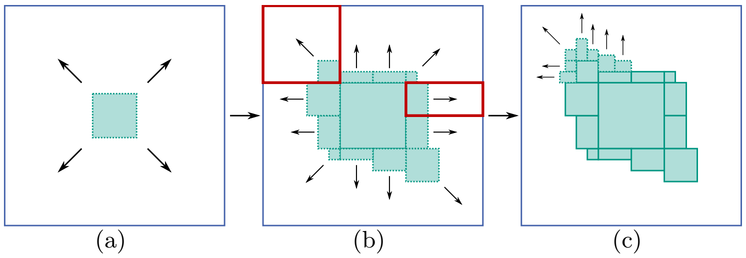


Fig. 6: Illustration of a single candidate (blue rectangle) refinement

tree in which the inner nodes are tests on the data and each leaf is labeled with a category. To classify an individual input, one follows the path given by the tests in the CT and obtains the label of the reached leaf as an answer.

We use CTs to find a simple classification that separates the *valid* from the *invalid* samples. Fig. 5a shows such a CT for our example `himmilbeau` function. Note that all tests in the CT are comparisons between a variable and a constant. From this CT, we can extract representations for the category *valid* by enumerating all paths from the root to valid leaves and collect (i.e. conjoin) all conditions (resp. their negation for negative edges). Due to the choice of simple comparisons with constants for tests, the result can be expressed as bounds on the input variables, which describes a set of interval regions. Fig. 5b shows the (simplified) precondition candidates extracted from Fig. 5a.

4.2 Refining Candidates by Growing Regions

Heuristics are applied when training CTs, and the classification has only been obtained from a set of few random samples. It is hence very likely that the candidates still contain inputs for which the desired postcondition does not hold. They need to be processed to obtain valid preconditions.

Fig. 6 illustrates our first candidate refinement process. The outer blue square represents the initial candidate. The verification backend is used to identify regions within it that verifiably are preconditions, shown as filled green rectangles in the figure. First, a small initial region in the center of the candidate is grown as much as possible without losing verifiability (Fig. 6a). When the maximal region has been found, additional precondition regions are inferred along the boundary of the region (Fig. 6b). To this end, extension candidates (two examples are shown as red rectangles) are identified as the largest possible regions to add in particular directions. The mentioned growing mechanism infers maximum regions within the extension candidates. For every added region, the extension process is repeated (Fig. 6c) until a maximum refinement depth has been reached.

Algorithm 2 shows the pseudocode procedure refineCandidate returning a verified precondition for a candidate *region*. The algorithm keeps a set M of extension candidates and searches for the largest verifiable region inside each extension candidate using binScaleSearch (binary search on interval regions)

Algorithm 2 Candidate Refinement

```
1: given arithmetic expression $expr$, postcondition $post$, binary search depth $d$
2: procedure REFINECANDIDATE($region$)
3:     $result \leftarrow \emptyset$
4:     $M \leftarrow \{(\textsc{center}(region), region)\}$
5:     while $M \neq \emptyset$ do
6:         choose $(min, max) \in M$ and remove
7:         $verified \leftarrow \textsc{binScaleSearch}(min, max)$
8:         if $verified \neq \emptyset$ then
9:             $result \leftarrow result \cup \{verified\}$
10:            $M \leftarrow M \cup \textsc{genExtensionCandidates}(verified, max)$
11:    return $result$
```

which invokes the verification backend. The procedure CENTER computes the center of a region used as the starting point for growing an initial solution, and GENEXTENSIONCANDIDATES produces new extensions candidates (in form of min/max pairs of regions) to be explored.

In the implementation, the set M is realized as a priority queue favoring potential additions far from the original candidate's border that can thus grow easily, and the number of iterations is bounded by a configurable parameter.

4.3 Refining Candidates by Recursive Subdivision

Instead of this refinement approach for precondition candidates, the subdivision technique from Sec. 3 can alternatively also be applied to obtain valid preconditions from candidates. The candidate production using a CT then serves as a first step narrowing an initial search region to a smaller region in which subdivision can operate productively, in particular because a finer mesh can be applied on the interesting regions, which is better for verification with the backend verifier.

5 Evaluation

Implementation We implemented both precondition inference approaches in the open-source tool Daisy [10], building on the static range and error analyses that Daisy provides. In particular, we use Daisy's interval analysis for computing real-valued ranges and affine arithmetic for computing roundoff error bounds. We use the `DecisionTree` class from the Smile library [1] for classification tree learning. Empirically, we have identified the following default parameters that produce good results on the benchmarks on average, while not being prohibitive for larger benchmarks: we limit the maximum depth for classification tree learning to 8 and the depth for binary search during refinement in the classification tree approach to 10. When combining classification tree learning with subdivision, we limit the decision tree depth at 12. We use 8192 samples for classification tree learning and 16 samples per subdivided region for our subdivision approach.

Benchmarks We evaluate our precondition inference approaches on benchmarks from the benchmark suite FPBench [8] that is widely used in the floating-point research community. Each benchmark consists of an arithmetic expression and typically comes with a precondition specifying the input domain of the expression. For a few benchmarks where no input domain is given, we add one manually. For our evaluation, we require a postcondition to be given that specifies a target error bound or a range. Since these are not provided by FPBench as-is, we generate them for our experiments as follows. We compute error bounds and result ranges based on the existing original input domains as specified in FPBench, and use these as two separate target postconditions. We exclude benchmarks for which Daisy is not able to compute errors or ranges, e.g. because they contain conditional statements. In total, we generate a set of 99 benchmarks with postconditions specifying an error bound, and a separate set of 99 benchmarks with postconditions specifying a target range, with the following dimensionalities:

dimension	1	2	3	4	6	8	9
# benchmarks	33	29	16	4	12	1	4

Baseline In the absence of existing tools for floating-point precondition inference or the ground truth[5], we compare the preconditions inferred by our approaches against the original preconditions specified in FPBench. Indeed, the original precondition from FPBench is—by construction—a valid precondition.

We measure the quality of an inferred precondition as a relative volume, i.e. the ratio of the volume of the generated precondition over the volume of the original precondition. A relative volume greater than one is obtained if the original domain specification is strong and the approaches discover valid preconditions beyond the original specification. For many benchmarks, however, obtaining a relative volume close to one is close to the optimal result. (Measuring the absolute volumes is not meaningful as they are highly benchmark dependent.)

Setup Our techniques rely on an initial search area provided by the user. While it may be convenient if our algorithms considered an unbounded initial space, i.e. all possible floating-point values, this is practically infeasible. The valid precondition typically covers only a very small part of this 'unbounded' domain, and it would thus be computationally very expensive to search for.

For our evaluation, we consider two sets of initial search areas: We use the original domain specified in FPBench *scaled* uniformly around their centers to contain 100 times the original volume, and we use a large *fixed* initial domain for all benchmarks bounding all input arguments in $[-10^8, 10^8]$. For both initial search areas, it is unlikely that the entire area would be a valid precondition.

Comparison of Approaches Simply comparing the relative volume of the preconditions does not consider that each approach would be able to produce bigger preconditions by investing more computational effort. Conversely, the running

[5] The exact ground truth would be highly discontinuous, and would require sampling of all floating-point inputs, which is infeasible for double precision.

precondition:	error			range		
	TO	fail	best	TO	fail	best
scaled search area						
subdivision	14	2	67	10	2	56
tree refinem.	6	3	22	9	3	24
hybrid	6	2	11	5	9	23
fixed search area						
subdivision	333	57	22	312	80	8
tree refinem.	344	73	4	319	81	4
hybrid	344	56	8	320	79	3

Fig. 7: Summary statistics

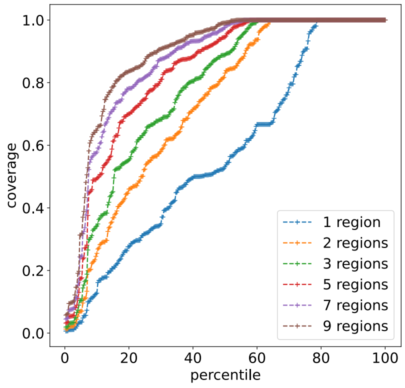


Fig. 8: Cactus plot evaluating the precondition optimization

times cannot be compared in isolation. Thus, we compare the relative volumes of generated preconditions per invested time[6]. We use a timeout of 20 minutes for each benchmark and parameter setting.

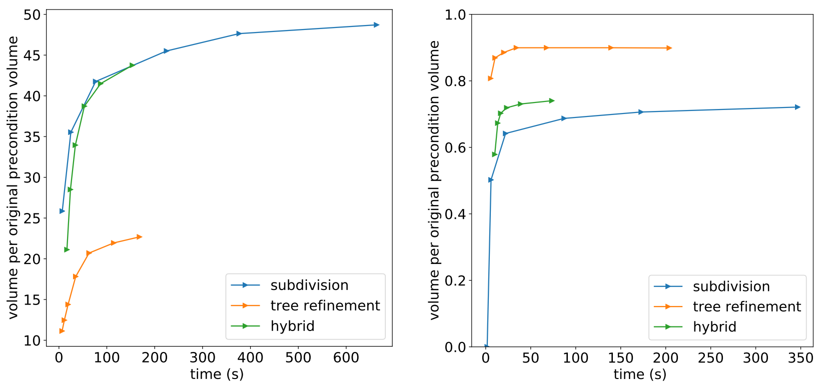
We consider our effectively three approaches: *subdivision*, *tree refinement* (with growing candidates), and tree refinement with subdivision, that we call *hybrid* for the sake of this evaluation. For this comparison, we initially do not use the precondition optimization from Sec. 3.2, and evaluate it separately. We observe that for the subdivision and the hybrid approach, the maximum depth of the subdivision tree significantly affects the running time of the algorithm. For the tree refinement, the most relevant parameter is the number of refinement candidates considered for the growing-based refinement. We thus vary these parameters and keep all others to the default values given in Sec. 5. In total, we run 3762 experiments using the scaled and 1782 experiments using the fixed search area.

Fig. 7 summarizes our results. 'TO' counts the number of times an individual run timed out. 'Fail' means that no precondition was found by a search strategy for any of the tested parameters. 'Best' counts the number of benchmarks for which an approach was able to find the best (weakest) precondition (with any parameter setting); when the numbers do not add up to 99, it is due to ties.

Clearly, our precondition inference is more effective for the scaled search area benchmarks; it is able to find preconditions in nearly all runs. However, it is able to find some preconditions even for the very large area, where the verifiable regions are often vanishingly small. Also, we observe that no one approach is universally better than the others, as each is best on some set of benchmarks.

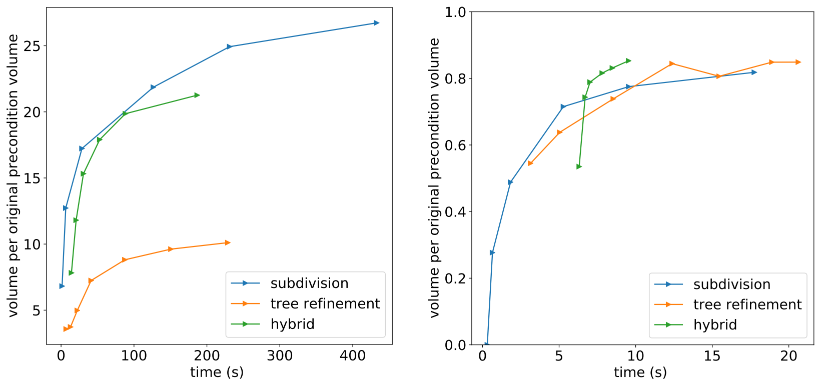
Fig. 9 visualizes the relative volumes of generated preconditions by the different approaches per running time of the algorithm, for benchmarks where the postconditions bound the roundoff error and for the scaled input search areas. Each point corresponds to one parameter setting. Fig. 9a averages over all benchmarks, whereas Fig. 9b averages only over benchmarks where the gener-

[6] We ran all experiments on a Mac mini with an 6-core Intel i5 processor at 3 GHz with 16 GB RAM running macOS Catalina.



(a) Average over all benchmarks (b) Average for small preconditions

Fig. 9: Comparison of approaches without optimization: average relative volume per time (seconds) for error postconditions and 100x search area.



(a) Average over all benchmarks (b) Average for small preconditions

Fig. 10: Comparison of approaches without optimization: average relative volume per time (seconds) for range postconditions and 100x search area.

ated precondition was small, i.e. at most 1.2 times the original precondition. We show the analogous plots for the range postconditions in Fig. 10.

We observe that averaged over all benchmarks, the subdivision and hybrid approaches perform significantly better than the tree refinement approach. In fact, our techniques are able to identify preconditions that are, on average, significantly larger than the original precondition. If we consider only those 33 benchmarks, where only a relative small precondition was generated, we see that tree refinement shows the, on average, best benefit. For our range benchmarks (Fig. 10), we observed on average a slight benefit of the hybrid approach for small preconditions. Note that even when 'small' preconditions are generated, they nearly cover the entire input search area, i.e. our precondition inference is able to recover most of the original preconditions.

Precondition Optimization Finally, we evaluate the effectiveness of the precondition optimization on the subdivision and hybrid approach (we have not observed the optimization to be particularly useful for tree refinement). For this evaluation, we fix a particular parameter setting that achieves a good trade-off between relative volume of inferred preconditions and running time of inference. Then we vary the number of target regions that the optimization should produce. On average, the preconditions generated for this experiment consisted of 120 distinct regions *before* optimization. For each run, we compute the *coverage* of the optimized precondition, i.e. the ratio of the optimized over the non-optimized inferred precondition. Fig. 8 visualizes the results of this experiment as a cactus plot where we sort the runs by coverage. For example, the value 0.27 for 1 region at the 20th percentile means that in 80% of the runs, the coverage of the optimized precondition was at least 0.27. As expected, the more regions are allowed, the better the coverage of the optimized preconditions becomes. Overall, we see that our inference with optimization is able to generate relatively simple preconditions (i.e. with just a few regions) in reasonable time that nonetheless cover large parts of the verifiable area for many of the benchmarks.

Case Study We demonstrate the benefits of our precondition inference on a practical problem that inspired this work. We consider the 9-dimensional function to calculate the scalar triple product $\alpha \cdot (\beta \times \gamma)$ of three 3-dimensional vectors $\alpha, \beta, \gamma \in \mathbb{R}^3$, based on the requirements of an assumed use case: each parameter will be within a range of $[-1337, 1337]$, and we require the error of the result to be at most $3 \cdot 10^{-6}$. This use case arose in a convex hull algorithm for scientific computing in material sciences.

Running the recursive subdivision approach for this expression with a subdivision depth of 14 and 262144 samples yields the following results: In roughly 13 minutes, the approach produces a precondition that covers about 67 percent of the search area and consists of 4608 individual intervals. In another 112 seconds, the optimization algorithm produces a precondition consisting of only two intervals which together cover 51% of the verified area and 34% of the search area. Using this optimized precondition, we can create a hybrid implementation of the original function, which decides whether to use a (exact) rational or floating-point version dynamically. Even with the added overhead from checking the precondition, the required runtime reduces from $17.13s$ for a purely rational implementation to $10.77s$ for the hybrid implementation for running the function 100000 times with random inputs from the input space. A similar speedup can be observed when using a higher precision floating-point implementation instead of an exact rational implementation in case the precondition does not hold.

6 Related Work

The precondition synthesis approaches presented in this work rely on state-of-the-art floating-point verification and analysis tools to verify precondition candidates and guarantee their soundness. While we have used the Daisy framework

[10] as a verification backend, any tool able to calculate sound bounds for errors or result ranges of floating-point functions could be used instead: Fluctuat [14], Gappa [11], FPTaylor [29], Real2Float [21] and PRECiSA [22].

We are not aware of an existing technique that can generate sound preconditions for floating-point functions. The closest related techniques are optimizations that identify certain parts of the input domain, for which a rewriting of the input program results in a smaller roundoff error [26,32,30]. These rewritings are based on real-valued identities, leveraging the fact that floating-point arithmetic is e.g. not associative, or polynomial approximations. The split of the input domain can be viewed as a kind of precondition, however, the goal and guarantees provided are very different. The aim is to identify and repair large roundoff errors, whereas our approach tries to identify the input domain with reasonable errors. Furthermore, all of the techniques rely on dynamic analysis and thus do not provide soundness guarantees.

Dynamic analysis is frequently being used to estimate the magnitude of roundoff errors [3], and several works have developed a targeted search towards inputs that cause particularly large errors [31,6,33], in order to identify worst-case errors. Our precondition inference combines dynamic and static analysis in a novel way in that the dynamic analysis serves a pre-processing step to explore the input domain. As such, the goal of our dynamic analysis is different from existing ones, as we want it to explore the input domain evenly, instead of focusing on a (possibly small) part of the input domain with large errors.

One possible use of our inferred preconditions is to be able to generate implementations that choose an efficient floating-point precision whenever possible, and otherwise use some 'safe' higher precision. In that, our approach is related to mixed-precision tuning techniques that mostly focus on implementations that mix single, double and quad floating-point precision. Some of these use dynamic analysis to estimate errors and thus do not provide sound guarantees [25,19,17,15], and others use static analysis with accuracy guarantees, but less scalability [5,9]. Mixed-precision tuning generally works well when the target error bounds are close to the error bounds of uniform-precision implementations [9,27]. We consider mixed-precision tuning complementary to our precondition inference; for instance, preconditions generated by our approaches could be used as a starting-point for mixed-precision tuning.

7 Conclusion

We have presented the first precondition inference techniques from floating-point accuracy and range postconditions, using a combination of dynamic and static analysis. Each of the three approaches that we explored generate good results from reasonably sized initial search areas with acceptable computational effort and have different strengths and weaknesses; neither approach is universally better than the others. One of the main challenges for future work is to improve the identification of preconditions when the initial search areas are very large, which we have identified as a particular challenge.

References

1. Smile - Statistical Machine Intelligence and Learning Engine, https://haifengl.github.io/
2. The GNU MPFR Library (2020), https://www.mpfr.org/
3. Benz, F., Hildebrandt, A., Hack, S.: A Dynamic Program Analysis to Find Floating-Point Accuracy Problems. In: Programming Language Design and Implementation (PLDI) (2012). https://doi.org/10.1145/2254064.2254118
4. Breiman, L., Friedman, J., Stone, C.J., Olshen, R.A.: Classification and Regression Trees. CRC press (1984)
5. Chiang, W.F., Baranowski, M., Briggs, I., Solovyev, A., Gopalakrishnan, G., Rakamarić, Z.: Rigorous Floating-Point Mixed-Precision Tuning. In: Principles of Programming Languages (POPL) (2017). https://doi.org/10.1145/3009837.3009846
6. Chiang, W., Gopalakrishnan, G., Rakamaric, Z., Solovyev, A.: Efficient Search for Inputs Causing High Floating-Point Errors. In: Principles and Practice of Parallel Programming (PPoPP) (2014). https://doi.org/10.1145/2555243.2555265
7. Claessen, K., Smallbone, N., Hughes, J.: QuickSpec: Guessing Formal Specifications Using Testing. In: Tests and Proofs (2010). https://doi.org/10.1007/978-3-642-13977-2_3
8. 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 (2017). https://doi.org/10.1007/978-3-319-54292-8_6
9. Darulova, E., Horn, E., Sharma, S.: Sound Mixed-Precision Optimization with Rewriting. In: International Conference on Cyber-Physical Systems (ICCPS) (2018). https://doi.org/10.1109/ICCPS.2018.00028
10. Darulova, E., Izycheva, A., Nasir, F., Ritter, F., Becker, H., Bastian, R.: Daisy - Framework for Analysis and Optimization of Numerical Programs (Tool Paper). In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (2018). https://doi.org/10.1007/978-3-319-89960-2_15
11. Daumas, M., Melquiond, G.: Certification of bounds on expressions involving rounded operators. ACM Transactions on Mathematical Software **37**(1) (2010). https://doi.org/10.1145/1644001.1644003
12. Dijkstra, E.W.: Guarded Commands, Nondeterminacy and Formal Derivation of Programs. Communications of the ACM **18**(8) (1975). https://doi.org/10.1145/360933.360975
13. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon System for Dynamic Detection of Likely Invariants. Science of Computer Programming **69**(1-3) (2007). https://doi.org/10.1016/j.scico.2007.01.015
14. Goubault, E., Putot, S.: Static Analysis of Finite Precision Computations. In: Verification, Model Checking, and Abstract Interpretation (VMCAI) (2011). https://doi.org/10.1007/978-3-642-18275-4_17
15. Guo, H., Rubio-González, C.: Exploiting Community Structure for Floating-Point Precision Tuning. In: International Symposium on Software Testing and Analysis (ISSTA) (2018). https://doi.org/10.1145/3213846.3213862
16. Himmelblau, D.M., Clark, B.J., Eichberg, M.: Applied Nonlinear Programming. McGraw-Hill (1972)
17. Ho, N., Manogaran, E., Wong, W., Anoosheh, A.: Efficient Floating Point Precision Tuning for Approximate Computing. In: Asia

and South Pacific Design Automation Conference (ASP-DAC) (2017). https://doi.org/10.1109/ASPDAC.2017.7858297

18. IEEE Computer Society: IEEE Standard for Floating-Point Arithmetic. IEEE Std 754-2019 (Revision of IEEE 754-2008) (2019). https://doi.org/10.1109/IEEESTD.2019.8766229
19. Lam, M.O., Vanderbruggen, T., Menon, H., Schordan, M.: Tool Integration for Source-Level Mixed Precision. In: Workshop on Software Correctness for HPC Applications (Correctness) (2019). https://doi.org/10.1109/Correctness49594.2019.00009
20. Loh, E., Walster, G.W.: Rump's Example Revisited. Reliable Computing **8**(3) (2002). https://doi.org/10.1023/A:1015569431383
21. Magron, V., Constantinides, G., Donaldson, A.: Certified Roundoff Error Bounds Using Semidefinite Programming. ACM Transactions on Mathematical Software **43**(4) (2017). https://doi.org/10.1145/3015465
22. Moscato, M., Titolo, L., Dutle, A., Muñoz, C.A.: Automatic Estimation of Verified Floating-Point Round-Off Errors via Static Analysis. In: Computer Safety, Reliability, and Security (SAFECOMP) (2017). https://doi.org/10.1007/978-3-319-66266-4_14
23. Moy, Y.: Sufficient Preconditions for Modular Assertion Checking. In: Verification, Model Checking, and Abstract Interpretation (VMCAI) (2008). https://doi.org/10.1007/978-3-540-78163-9_18
24. Muller, J.M., Brunie, N., de Dinechin, F., Jeannerod, C.P., Joldes, M., Lefèvre, V., Melquiond, G., Revol, N., Torres, S.: Handbook of Floating-Point Arithmetic. Springer International Publishing (2018). https://doi.org/10.1007/978-3-319-76526-6
25. Nathan, R., Naeimi, H., Sorin, D.J., Sun, X.: Profile-Driven Automated Mixed Precision. CoRR **abs/1606.00251** (2016), http://arxiv.org/abs/1606.00251
26. Panchekha, P., Sanchez-Stern, A., Wilcox, J.R., Tatlock, Z.: Automatically Improving Accuracy for Floating Point Expressions. In: Programming Language Design and Implementation (PLDI) (2015). https://doi.org/10.1145/2737924.2737959
27. Rabe, R., Izycheva, A., Darulova, E.: Regime Inference for Sound Floating-Point Optimizations. ACM Trans. Embed. Comput. Syst. (EMSOFT) **20**(5s) (2021). https://doi.org/10.1145/3477012
28. Seghir, M.N., Kroening, D.: Counterexample-Guided Precondition Inference. In: Programming Languages and Systems (ESOP) (2013). https://doi.org/10.1007/978-3-642-37036-6_25
29. Solovyev, A., Baranowski, M.S., Briggs, I., Jacobsen, C., Rakamarić, Z., Gopalakrishnan, G.: Rigorous Estimation of Floating-Point Round-Off Errors with Symbolic Taylor Expansions. ACM Transactions on Programming Languages and Systems **41**(1) (2018). https://doi.org/10.1145/3230733
30. Wang, X., Wang, H., Su, Z., Tang, E., Chen, X., Shen, W., Chen, Z., Wang, L., Zhang, X., Li, X.: Global Optimization of Numerical Programs via Prioritized Stochastic Algebraic Transformations. In: International Conference on Software Engineering (ICSE) (2019). https://doi.org/10.1109/ICSE.2019.00116
31. Xia, Y., Guo, S., Hao, J., Liu, D., Xu, J.: Error Detection of Arithmetic Expressions. The Journal of Supercomputing (2020). https://doi.org/10.1007/s11227-020-03469-7
32. Yi, X., Chen, L., Mao, X., Ji, T.: Efficient Automated Repair of High Floating-Point Errors in Numerical Libraries. Proceedings of the ACM on Programming Languages **3**(POPL) (2019). https://doi.org/10.1145/3290369

33. 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 Software Engineering (ICSE) (2015). https://doi.org/10.1109/ICSE.2015.70

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.