Making Interval Arithmetic Robust to Overflow

Oliver Flatt
Paul G. Allen School
University of Washington
oflatt@cs.washington.edu

Pavel Panchekha
Kahlert School of Computing
University of Utah
pavpan@cs.utah.edu

***Abstract*—In theory, interval arithmetic at high precision can compute mathematical expressions to any required accuracy. An arbitrary precision library like MPFR can thus be used to evaluate real-valued expressions. In practice, however, MPFR's maximum representable exponent means some inputs cannot be evaluated to the required accuracy. This paper introduces *movability flags*, which soundly detect the majority of these inputs. Movability flags are set when overflow occurs and track whether recomputing at higher precision could possibly result in narrow intervals. In our tests on 481 expressions, movability flags detect 81.0% of unsamplable inputs. Compared to Mathematica, our movability-flag-enhanced interval arithmetic library resolves 60.3% more challenging inputs, returns 7.6× fewer completely indeterminate results, and avoids 64 cases of fatal error.**

***Index Terms*—Interval arithmetic, overflow, computable reals, multiple precision arithmetic**

1. Overview

Consider computing $x^y/(x^y+2)$ for various inputs (x, y) to an accuracy of two decimal digits.[1] Using two-digit decimal interval arithmetic on input $(3.0, 1.1)$ results in the interval $[0.61, 0.65]$. The result represents a range because of rounding error; for example, $3.0^{1.1}$ is not exactly representable with two-digit decimals so must return $[3.3, 3.4]$. If our ultimate goal is to compute the result to two decimal digits, this answer is not sufficiently precise; any of 0.61, 0.62, 0.63, 0.64, or 0.65 could be the correct answer. *Recomputing* the the result in higher precision can help. For example, using four-digit decimal intervals, $[3.0, 3.0]^{[1.1,1.1]} = [3.348, 3.349]$ and the final result is therefore the much narrower interval $[0.6259, 0.6263]$. Importantly, both endpoints round to two-digit decimal value 0.63, meaning that the true value does too. Even higher precisions could be used if four-digit intervals don't produce a narrow-enough result.

Yet now consider the extreme input $(x, y) = (10^{10}, 10^{10})$, which might come from an automated tool like error estimation, stability analysis, or model checking. On this input, x^y is huge, overflowing to infinity in the MPFR arbitrary-precision library. So x^y must return $[\Omega_p, \infty]$, where Ω_p is the largest finite value in precision p. The final resut is thus $[\Omega_p, \infty]/[\Omega_p, \infty] = [0, \infty]$, too wide to be useful at any precision p. Automated tools will typically attempt to recompute at higher and higher precisions, but these recomutations just waste time because this error occurs at any precision.

Movability flags detect futile recomputation. When x^y overflows, for example, the infinite endpoint in $[\Omega_p, \infty]$ is marked "immovable", because it will evaluate to ∞ and *any* higher precision $q > p$.[2] We write the movability flag with an exclamation mark: $[\Omega_p, \infty!]$. These movability flags can be soundly propagated through interval computations: $[\Omega, \infty!] + 2$ also has an immovable infinity, and the final result $[!0, \infty!]$ has two immovable endpoints—because the left endpoint is computed by dividing by immovable infinity, while the right endpoint has an immovable infinity in the numerator. Because both endpoints are immovable, recomputation won't help and an error can be printed immediately.[3]

Importantly, movability flags are sound: an interval can only become immovable if recomputation at *any* higher precision yields the same result. Movability flags thus still allow recomputation where it is necessary, and avoid warning about benign overflows like in $1/e^x$. While movability flags can't detect all cases of futile recomputation—the problem is likely undecidable [1]—they do detect the vast majority, thereby making tools that use interval arithmetic faster and more robust.

2. Movable and Immovable Endpoints

Let S_p be the set of real numbers representable in precision p using MPFR, and let $R_p^\downarrow$ and $R_p^\uparrow$ be the rounding functions (up and down) from $\mathbb{R}$ to S_p. An interval $[a, b] \in S_p^2$ then represents the set $\{x \in \mathbb{R} \mid a \le x \le b\}$ and functions $f : \mathbb{R} \to \mathbb{R}$ can be extended to functions $f_p : S_p^2 \to S_p^2$ such that if $x \in I$ then $f(x) \in f_p(I)$.

Movability flags augment each interval with a boolean flag at each endpoint; we say the endpoint is *immovable* when

1. For ease of exposition, this section uses decimal floating-point, though MPFR and Rival in fact use binary floating point.

2. The left endpoint Ω_p is not marked immovable, because slightly larger "largest" numbers are available at higher precisions.

3. Or a warning could be printed, or the input point skipped, or some other application-appropriate action.

its movability flag is set. An immovable endpoint doesn't change if recomputed at a higher precision, formalized as a notion of refinement:

Definition 1. *One interval* refines *another, written* $[a', b']_p \prec [a, b]_q$*, when* $b' = b$ *and both are immovable or when* $b' \leq b$ *and* b *is movable; and likewise* $a' = a$ *and both are immovable or when* $a' \geq a$ *and* a *is movable.*

An interval library with movability flags then guarantees that each function is isotonic with respect to refinement:

Property 2. $f_q(I'_1, \ldots, I'_n) \prec f_p(I_1, \ldots, I_n)$ *when* $q > p$ *and* $I'_i \prec I_i$*, for all functions* f*.*

If $f_p(I, J)$ has an immovable endpoint, $f_q(I', J')$ must share that immovable endpoint; if both endpoints are immovable, the interval has only one refinement so recomputation can be skipped. The remainder of this section describes how to guarantee Property 2. Because immovable endpoints strictly restrict refinement compared to movable ones, that largely means knowing when an interval can safely be marked immovable.

2.1. Constants and Overflow

Point intervals $[x, x]$, like for numeric constants 2 or 0.5, only refine to themselves, so both endpoints can be marked immovable. Inexact constants like π, however, yield narrower intervals at higher precisions so have movable endpoints.

Computations that overflow also result in immovable endpoints. For example, in the MPFR library, exponents are bounded by 2^H for some H, so any real number equal to 2^{2^H+1} or larger rounds up to $+\infty$ at any precision. Rapidly-growing functions like exp, exp2, and pow can thus set the right movability flag for particularly large inputs:

Lemma 1. *The output of* $\exp_p([a, b])$ *has an immovable right endpoint if either a)* $a \geq (2^H + 1)(\log 2)$*, or b)* $b \geq (2^H + 1)(\log 2)$ *and also* b *is immovable.*

Proof. If $a \geq (2^H + 1)(\log 2)$, all points in the input interval maps to values larger than 2^{2^H+1}. The right endpoint then rounds up to $+\infty$ no matter the precision p. It is therefore sound to mark the right endpoint immovable.

The case where only $b \geq (2^H + 1)(\log 2)$ is more complex. The right endpoint is equal to infinity, but it can only have its movability flag set if it would also equal infinity for all refinements of $[a, b]$. If $a < (2^H + 1)(\log 2)$, this is only true if b is immovable. □

Note that similar reasoning *does not* apply to the left endpoint: arbitrarily-small numbers can be represented by MPFR as denormal values at a high-enough precision.

2.2. One-Argument Functions

Functions propagate movability flags from their inputs to their output. Consider a monotonic function $f(x)$ for illustration. Its behavior on intervals is particularly simple:

$$f_p([a, b]) = [R^{\downarrow}_p(f(a)), R^{\uparrow}_p(f(b))].$$

Now suppose a is an immovable endpoint; the left endpoint in any higher precision q will thus be $R^{\downarrow}_q(f(a))$. These will be the same only if the rounding behavior is the same, which occurs when $f(a)$ is exact at precision p. Formally:

$$\begin{aligned} f_q([!a, b']) &= f_q([a, b']) = [R^{\downarrow}_q(f(a)), R^{\uparrow}_q(f(b'))] \\ &= [f(a), R^{\uparrow}_q(f(b'))] \prec [f(a), R^{\uparrow}_p(f(b))] = f_p([!a, b]) \end{aligned}$$

To generalize, for monotonic functions, exact computations on immovable endpoints result in immovable endpoints. For example, in $\mathsf{sqrt}_p([0, 4!]) = [0, 2!]$, 2 is immovable because the square root of 4 is exact, but in $\mathsf{sqrt}_p([0, 2!]) = [0, 1.414\ldots]$, the right endpoint is movable because $\sqrt{2}$ is not exact. MPFR reports whether an operation is exact, so this rule is implementable.

Non-monotonic functions are more complex. They can be thought of as computing f on *critical points* a' and b':

$$f_p([a, b]) = [R^{\downarrow}_p(f(a')), R^{\uparrow}_p(f(b'))] \text{ where } a', b' \in [a, b]$$

If we think of the computation of a' and b' as an intermediate step in f_p, the rule for endpoint movability becomes:

Lemma 2. *Suppose* $f_p([a, b])$ *computes its left endpoint via* $f(a')$*, which is also exact in precision* p*. That endpoint may be marked immovable if either 1)* $a' = a$ *and* a *is immovable; 2)* $a' = b$ *and* b *is immovable; or 3)* a *and* b *are both immovable and* a' *is computed exactly. The right endpoint is analogous.*

Proof. Consider a refinement $I \prec [a, b]$. First note that $a' \in I$: if condition (1) or (2) holds, a' is equal to an endpoint preserved by refinement, while if condition (3) holds, $I = [a, b]$. Since a' minimizes f over the interval $[a, b]$ and $a' \in I \prec [a, b]$, a' also minimizes f over I. Since $f(a')$ bounds f from below over I, therefore so does $R^{\downarrow}_q(f(a'))$ for any precision $q > p$. Since $a' \in [a, b]$ that means $R^{\downarrow}_q(f(a'))$ is the left endpoint of $f_q(I)$. If, furthermore, $f(a')$ is exact in precision p, then $R^{\downarrow}_q(f_q(a')) = R^{\downarrow}_p(f(a'))$. Therefore $f_p([a, b])$ and $f_q(I)$ have the same left endpoint, meaning that it satisfies Property 2. □

Computing these critical points is easy for most functions; For example, for $\mathsf{fabs}([a, b])$, the left critical point is a, b, or 0, and the right critical point is a or b. Also note that Lemma 2 does not mention the precision of a'; this means that it's valid to compute it symbolically. For example, cos has a minimum of -1 at π. Thinking of π symbolically thus allows marking the left endpoint of $\cos([!3, 4!])$ immovable by condition (3). Most importantly, Computations on infinities are exact, so Lemma 2 preserves these typicaly products of overflow.

2.3. Many-Argument Functions

Most many-argument functions can be implemented via the following generalization of Lemma 2:

Lemma 3. *Suppose* $f_p([a_1, b_1], \ldots, [a_n, b_n])$ *computes its left endpoint via* $f(a'_1, \ldots, a'_n)$*, which is also exact in*

precision p. That endpoint may be marked immovable if, for all i, either 1) $a'_i = a_i$ and a_i is immovable; 2) $a'_i = b_i$ and b_i is immovable; or 3) a_i and b_i are both immovable and a'_i is computed exactly. The analogous applies to f_p's right endpoint.

The proof is entirely analogous to Lemma 2. However, the "for all i" clause is too conservative for some common functions. Addition is one example. Consider $[1, \infty!] +_p [1, 2]$; this falls in none of the cases of Lemma 3, but the right output endpoint, ∞, should still be immovable because anything plus infinity is infinity. Similar reasoning applies to hypot_p and to multiplication by 0. These functions thus require special-cased implementations to propagate movability flags in as many cases as possible.

A more complex special case is multiplication by infinity. For example, $[0, 1] \times_p [!1, +\infty!] = [0, \infty!]$ and $[-1, 0] \times_p [!1, +\infty!] = [! - \infty, 0]$. But since both refine $[-1, 1]$, we have $[-1, 1] \times_p [!1, +\infty!] = [-\infty, +\infty]$, where the output interval must be movable. More generally, handling infinite values in multiplication requires knowing the sign of the input interval:

Lemma 4. *Let $c = a'_i \times_p b'_i$ be an endpoint of an interval returned by multiplication. The endpoint may be marked immovable if: 1) both a'_i and b'_i are immovable and c is computed exactly; or, 2) a'_i is zero and immovable (or likewise for b'_i); or, 3) a'_i is infinite and immovable and $[b_1, b_2]$ does not contain zero (or likewise for b'_i).*

Proof. The first case just restates Lemma 2. In the second case, a'_i is immovable, so $a'_i = 0$ is in any refinement. Since $0 \times b'_i = 0$ for any b'_i, the result is immovable. Finally, in the third case, all elements of $[b_1, b_2]$ have the same sign. So since a'_i is immovable, $a'_i \times b'_i$ has a fixed sign. Since a'_i is infinity, the resulting endpoint is the same in any refinement. □

Similar logic applies to the pow_p function with zero, unit, or infinite arguments; our implementation of pow_p only sets movability flags when the first argument is positive. These special cases seem obscure, but are in fact important because overflow frequently produces immovable special values like 0, 1, and infinity, which must then be propagated.

3. Stand-Alone Evaluation

We implement movability flags in Rival, a free software interval arithmetic library for Racket largely conforming to IEEE 1788, including the def and ill decorations. Code and documentation is available online at https://docs.racket-lang.org/rival/. We test Rival on the Herbie 1.4 benchmark suite, which contains 481 floating-point expressions from numerical methods textbooks, mathematics and physics papers, and surveys of open-source code. For each expression, we sample 8 256 random inputs and evaluate it at each input to double precision. The working precision starts at 80 bits and doubles on recomputation, capped at 10 240 bits. The key evaluation condition is the number of points that hit that cap without producing an immovable interval.

In total, movability flags detect 81.0% of such points and thus prevent most futile recomputations (Figure 1). These points come from 21 expressions, mostly ratios of exponentials. For example, the expq2 benchmark computes $\exp(x)/(\exp(x)-1)$; for inputs like $x = 10^{100}$ the numerator and denominator are both too large to be represented. Movability flags detect this at just 80 bits of precision.

The expq3 requires higher working precisions to produce an immovable interval. This benchmark computes:

$$\frac{\varepsilon \cdot \left(e^{(a+b)\cdot\varepsilon} - 1\right)}{\left(e^{a\cdot\varepsilon} - 1\right) \cdot \left(e^{b\cdot\varepsilon} - 1\right)}, \text{ with } -1 < \varepsilon < 1.$$

For $a = \varepsilon = 10^{-100}$ and $b = 10^{200}$, the numerator and denominator both have terms that overflow. However, at low precisions, the first term in the denominator rounds down to 0, making the denominator possibly zero and preventing movability flags from propagating to the output. Only at 2560 bits of precision is an immovable output interval produced. There are a couple of similar examples.

Movability flags do fail on some more challenging examples. For example, consider the exp-w benchmark, $e^{-w} \cdot \mathsf{pow}(\ell, e^w)$, when ℓ is negative and w large and positive. Here, $\mathsf{pow}(\ell, e^w)$ raises a negative number to a possibly-infinite power; movability flags are not set because our implementation of pow only sets movability flags when the first argument is positive. Movability flags also fail on the classic difficult-to-analyze function "Kahan's Monster":

require $y > 0$
let $z = |y - \sqrt{y^2+1}| - 1/(y + \sqrt{y^2+1}), w = z \cdot z$ **in**
if $w = 0$ **then** 1 **else** $(\exp w - 1)/w$

Here Rival cannnot rule out a division by zero in the else branch, so again cannot propagate movability flags. Despite these failures, the overall 81.0% success rate shows that movability flags detect the vast majority of futile recomputations.

4. Comparison with Mathematica

We also compared Rival to the `N` function in Mathematica 12.1.1. Mathematica's documentation suggests that, like Rival, `N` is intended to handle invalid inputs soundly and cut off recomputation in some cases of over- and under-flow.[4] We thus ran Mathematica on the exact same expressions *and inputs* as in Section 3. To do so, we set its precision limit, $MaxExtraPrecision [2], identically to Rival and ask it to evaluate the expressions We ask `N` to evaluate to 15.6 decimal digits of precision, the rough equivalent to double-precision accuracy. If Mathematica throws an error related to its precision limit, we consider that a failure to detect futile recomputation, while we generously assume that any warning indicates that Mathematica detected a possible futile recomputation. We also performed a variety of cross-checks to ensure that Mathematica and Rival perform the same computation and that Mathematica's errors and warnings

4. Of course, being a proprietary system, it's impossible to know with certainty how Mathematica's `N` works, and the documentation does not always clarify.

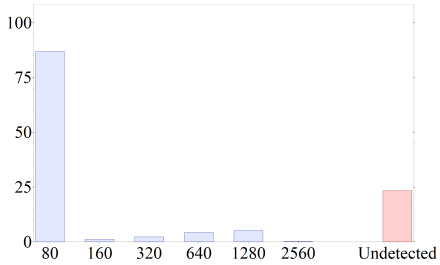

Figure 1. Unsamplable points and the precisions they were detected at. Only 19.0% continue recomputing up to the working precision cap. For all other points, Rival's movability flags produce a stuck interval.

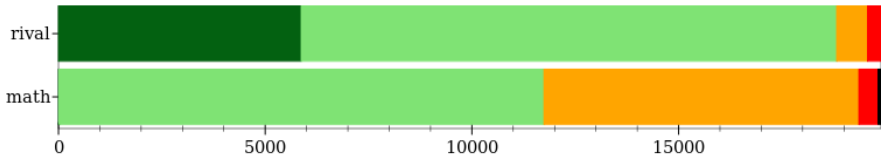

Figure 2. Rival versus Mathematica's `N` on 19895 hard inputs for the Herbie 1.4 benchmark suite. Rival achieves better results for each outcome threshold, including successfully sampled points (dark green), points proven invalid (light green), unsamplable points (orange), or points with unknown results (red). Black points cause Mathematica to run out of memory, crash, or freeze.

are treated as generously as possible. These cross-checks passed, except in two cases, which we investigated, reported to Wolfram support as bugs, and had those bugs confirmed. We focus in this section on "hard" inputs where either Rival or Mathematica detected a futile recomputation. Our results are plotted in Figure 2.

Of the 19895 hard input points, Mathematica fails to detect futile recomputations for 8151 while Rival fails on 1071 (7.6× better). Rival detects 5877 points not detected by Mathematica, while Mathematica detected only 26 points that Rival did not. In other words, Rival detects a near-superset of points detected by Mathematica.[5] Moreover, Rival's movability flags are sound, while Mathematica's raises warnings on 5830 inputs where Rival ultimately was able to compute a value. Mathematica also reaches its internal precision limit 1.4× more often than Rival.

In a few (64) cases, Mathematica takes over a second, seemingly in an infinite loop, or declares that insufficient memory is available.[6] We render these inputs in black in Figure 2. In 16 of these cases, the Mathematica process does not respond to shutdown requests or even the SIGTERM signal, and must be killed with SIGKILL. Finally, while not a primary metric for comparison, Rival is 9.87× faster than `N`. Even if Mathematica's timeouts, OOM, and crashes are subtracted from its runtime, Rival is still 9.03× faster. While it is difficult to know the root causes of these problems, Rival's movability flags clearly enable more robustness.

5. Discussion

Rival was written to evaluate mathematical expressions in the Herbie floating-point error estimation and repair tool [3]. Herbie originally used slow and unsound heuristics for this task; Herbie 1.4 [4] switched to Rival, which is faster and also sound. Rival has since been used by thousands of users.

Rival is structured as a traditional interval arithmetic library, with a strong emphasis on correctness. Most functions produce "tight" intervals in the sense of IEEE 1788; all functions correctly propagate the def and ill annotations. To ensure tight intervals, we split each function's domain into *monotonic regions* and computed critical points for each monotonic region on paper. For some functions, like fmod, this was challenging. We then implemented each interval function using those critical points. To catch errors, we deployed millions of random tests for soundness, weak completeness, and movability, which caught several additional typos and oversights, like sin internally rounding π incorrectly. As a result of this focus on correctness, we have high confidence that Rival intervals are tight for most functions and valid for all of them.

Movability flags were developed to speed up Rival. We found that futile recomputation frequently caused timeouts; lowering the maximum precision reduced timeouts but caused sampling failures. Movability flags grew out of an attempt to instead detect and skip futile recomputations, and ended up speeding up Rival by a factor of roughly 3×.

5. Without access to Mathematica's internals it's hard to say, but our best guess is that Mathematica simplifies these expressions before evaluating them, allowing it to succeed where Rival fails.

6. Our machine has 32GB of memory; Rival never runs out of memory or takes longer than 250 ms to per input.

Acknowledgements

We thank Zachary Tatlock, for guidance and valuable suggestions while preparing the paper. This work was supported by NSF award 2119939. This material is based upon work supported by the U.S. Department of Energy, under award number 10061193.

References

[1] H.-J. Boehm, "Towards an API for the real numbers," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 562–576. [Online]. Available: https://doi.org/10.1145/3385412.3386037

[2] Wolfram, "$MaxExtraPrecision—Wolfram Language documentation," 2020. [Online]. Available: https://reference.wolfram.com/language/ref/%24MaxExtraPrecision.html

[3] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock, "Automatically improving accuracy for floating point expressions," ser. PLDI, 2015.

[4] B. Saiki, O. Flatt, C. Nandi, P. Panchekha, and Z. Tatlock, "Combining precision tuning and rewriting," in *2021 IEEE 28th Symposium on Computer Arithmetic (ARITH)*, 2021.