




Correct approximation of IEEE 754 floating-point arithmetic for program verification

Roberto Bagnara^{1,2} · Abramo Bagnara² · Fabio Biselli^{2,3} · Michele Chiari^{2,4}  · Roberta Gori⁵

Accepted: 7 November 2021 / Published online: 22 February 2022
© The Author(s) 2022

Abstract

Verification of programs using floating-point arithmetic is challenging on several accounts. One of the difficulties of reasoning about such programs is due to the peculiarities of floating-point arithmetic: rounding errors, infinities, non-numeric objects (NaNs), signed zeroes, denormal numbers, different rounding modes, etc. One possibility to reason about floating-point arithmetic is to model a program computation path by means of a set of ternary constraints of the form $z = x \boxdot y$ and use constraint propagation techniques to infer new information on the variables' possible values. In this setting, we define and prove the correctness of algorithms to precisely bound the value of one of the variables x , y or z , starting from the bounds known for the other two. We do this for each of the operations and for each rounding mode defined by the IEEE 754 binary floating-point standard, even in the case the rounding mode in effect is only partially known. This is the first time that such so-called *filtering algorithms* are defined and their correctness is formally proved. This is an important slab for paving the way to formal verification of programs that use floating-point arithmetics.

✉ Roberto Bagnara
bagnara@cs.unipr.it

Abramo Bagnara
abramo.bagnara@bugseng.com

Fabio Biselli
fabio.biselli@bugseng.com

Michele Chiari
michele.chiari@polimi.it

Roberta Gori
gori@di.unipi.it

¹ Dipartimento di Scienze Matematiche, Fisiche e Informatiche, Università di Parma, Parma, Italy

² BUGSENG srl, Parma, Italy

³ Certus Software V & V Center, SIMULA Research Laboratory, Fornebu, Norway

⁴ Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Milan, Italy

⁵ Dipartimento di Informatica, Università di Pisa, Pisa, Italy

Keywords Constraint solving · Constraint propagation · Constraint satisfaction problem · Filtering algorithm · Floating point · Program verification · Symbolic execution

1 Introduction

Programs using floating-point numbers are notoriously difficult to reason about [33]. Many factors complicate the task:

1. compilers may transform the code in a way that does not preserve the semantics of floating-point computations;
2. floating-point formats are an implementation-defined aspect of most programming languages;
3. there are different, incompatible implementations of the operations for the same floating-point format;
4. mathematical libraries often come with little or no guarantee about what is actually computed;
5. programmers have a hard time predicting and avoiding phenomena caused by the limited range and precision of floating-point numbers (overflow, absorption, cancellation, underflow, etc.); moreover, devices that modern floating-point formats possess in order to support better handling of such phenomena (infinities, signed zeroes, denormal numbers, non-numeric objects a.k.a. NaNs) come with their share of issues;
6. rounding is a source of confusion in itself; moreover, there are several possible rounding modes and programs can change the rounding mode any time.

As a result of these difficulties, the verification of floating-point programs in industry relies, almost exclusively, on informal methods, mainly testing, or on the evaluation of the numerical accuracy of computations, which only allows to determine conservative (but often too loose) bounds on the propagated error [19].

The satisfactory formal treatment of programs engaging in floating-point computations requires an equally satisfactory solution to the difficulties summarized in the above enumeration. Progress has been made, but more remains to be done. Let us review each point:

1. Some compilers provide options to refrain from rearranging floating-point computations. When these are not available or cannot be used, the only possibility is to verify the generated machine code or some intermediate code whose semantics is guaranteed to be preserved by the compiler back-end.
2. Even though the used floating-point formats are implementation-defined aspects of, say, C and C++¹ the wide adoption of the IEEE 754 standard for binary floating-point arithmetic [24] has improved things considerably.
3. The IEEE 754 standard does provide some strong guarantees, e.g., that the results of individual additions, subtractions, multiplications, divisions and square roots are correctly rounded, that is, it is *as if* the results were computed in the reals and then rounded as per the rounding mode in effect. But it does not provide guarantees on the results of other operations and on other aspects, such as, e.g., when the underflow exception is signaled [17].²

¹This is not relevant if one analyzes machine or sufficiently low-level intermediate code.

²The indeterminacy described in [17] is present also in the 2008 edition of IEEE 754 [24].

4. A pragmatic, yet effective approach to support formal reasoning on commonly used implementation of mathematical functions has been recently proposed in [6]. The proposed techniques exploit the fact that the floating-point implementation of mathematical functions preserve, not completely but to a great extent, the piecewise monotonicity nature of the approximated functions over the reals.
5. A static analysis for detecting floating-point exceptions based on abstract interpretation has been presented in [32]. A few attempts at this task have been made using other techniques [7, 38] but, as we argue in Sections 1.4 and 5, they present precision and soundness issues.
6. Most verification approaches in the literature assume the round-to-nearest rounding mode [10], or over-approximate by always considering worst-case rounding modes [32]. Analyses based on SMT solvers [13] can treat each rounding mode precisely, but only if the rounding mode in use is known exactly. As we show in Section 5, some SMT solvers also suffer from soundness issues.

The contribution of this paper is in areas 5 and 6. In particular, concerning point 5, by defining and formally proving the correctness of constraint propagation algorithms for IEEE 754 arithmetic constraints, we enable the use of formal methods for a broad range of programs. Such methods, i.e., abstract interpretation and symbolic model checking, allow for proving that a number of generally unwanted phenomena (e.g., generation of NaNs and infinities, absorption, cancellation, instability, etc.) do not happen or, in case they do happen, allow the generation of a test vector to reproduce the issue. Regarding point 6, handling of all IEEE 754 rounding modes, and being resilient to uncertainty about the rounding mode in effect, is another original contribution of this paper.

While the round-to-nearest rounding mode is, by far, the most frequently used one, it must be taken into account that:

- the possibility of programmatically changing the rounding mode is granted by IEEE 754 and is offered by most of its implementations (e.g., in the C programming language, via the `fesetround()` standard function);
- such possibility is exploited by interval libraries and by numerical calculus algorithms (see, e.g., [35, 36]);
- setting the rounding mode to something different from round-to-nearest can be done by third parties in a way that was not anticipated by programmers: this may cause unwanted non-determinism in video games [20] and there is nothing preventing the abuse of this feature for more malicious ends, denial-of-service being only the least dangerous in the range of possibilities. Leaving malware aside, there are graphic and printer drivers and sound libraries that are known to change the rounding mode and may fail to set it back [37].

As a possible way of tackling the difficulties described until now, and enabling sound formal verification of floating-point computations, this paper introduces new algorithms for the propagation of arithmetic constraints over floating-point numbers. These algorithms are called *filtering algorithms* as their purpose is to prune the domains of possible variable values by *filtering out* those values that cannot be part of the solution of a system of constraints. Algorithms of this kind must be employed in *constraint solvers* that are required in several different areas, such as automated test-case generation, exception detection or the detection of subnormal computations. In this paper we propose fully detailed, provably correct filtering algorithms for floating-point constraints. Such algorithms handle all values, including symbolic values (NaNs, infinities and signed zeros), and rounding modes defined by IEEE 754. Note that filtering techniques used in solvers over the reals do not preserve

all solutions of constraints over floating-point numbers [30, 31], and therefore they cannot be used to prune floating-point variable domains reliably. This leads to the need of filtering algorithms such as those we hereby introduce.

The choice of the IEEE 754 Standard for floating-point numbers as the target representation for our algorithms is due to their ubiquity in modern computing platforms. Indeed, although some programming languages leave the floating-point format as an implementation-defined aspect, all widely-used hardware platforms —e.g., x86 [25] and ARM [3]—only implement the IEEE 754 Standard, while older formats are considered legacy.

Before defining our filtering algorithms in a detailed and formal way, we provide a more comprehensive context on the propagation of floating-point constraints and its practical applications (Sections 1.1 and 1.2), and justify their use in formal program analysis and verification (Section 1.3). We also give a more in-depth view of related work in Section 1.4, and clarify our contribution in Section 1.5.

1.1 From programs to floating-point constraints

Independently from the application, program analysis starts with parsing, the generation of an *abstract syntax tree* and the generation of various kinds of intermediate program representations. An important intermediate representation is called *three-address code* (TAC). In this representation, complex arithmetic expressions and assignments are decomposed into sequences of assignment instructions of the form

$$\text{result} := \text{operand}_1 \text{ operator } \text{operand}_2.$$

A further refinement is the computation of the *static single assignment form* (SSA) [2] whereby, labeling each assigned variable with a fresh name, assignments can be considered as if they were equality constraints. For example, the TAC form of the floating-point assignment $z := z * z + z$ is $t := z * z$; $z := t + z$, which in an SSA form becomes $t_1 := z_1 * z_1$; $z_2 := t_1 + z_1$. These, in turn, can be regarded as the conjunction of the constraints $t_1 = z_1 \boxtimes z_1$ and $z_2 = t_1 \boxplus z_1$, where by \boxtimes and \boxplus we denote the multiplication and addition operations on floating-point numbers, respectively. The Boolean comparison expressions that appear in the guards of if statements and loops can be translated into constraints similarly. This way, a C/C++ program translated into an SSA-based intermediate representation can be represented as a set of constraints on its variables. In particular, a constraint set arises from each execution path in the program. For this reason, this approach to program modeling can be viewed as *symbolic execution* [15, 26]. Constraints can be added or removed from such a set in order to obtain a constraint system that describes a particular behavior of the program (e.g., the execution of a certain instruction, the occurrence of an overflow in a computation, etc.). Once such a constraint system has been solved, the variable domains only contain values that cause the desired behavior. If one of the domains is empty, then that behavior can be ruled out. For more details on the symbolic execution of floating-point computations, we refer the reader to [4, 10].

1.2 Constraint propagation

Once constraints have been generated, they are amenable to *constraint propagation*: under this name goes any technique that entails considering a subset of the constraints at a time, explicitly removing elements from the set of values that are candidate to be assigned to the constrained variables. The values that can be removed are those that cannot possibly participate in a solution for the selected set of constraints. For instance, if a set of

floating-point constraints contains the constraint $x \sqcap x = x$, then any value outside the set $\{\text{NaN}, +0, 1, +\infty\}$ can be removed from further consideration. The degree up to which this removal can actually take place depends on the data-structure used to record the possible values for x , intervals and multi-intervals being typical choices for numerical constraints. For the example above, if intervals are used, the removal can only be partial (negative floating-point numbers are removed from the domain of x). With multi-intervals more precision is possible, but any approach based on multi-intervals must take measures to avoid combinatorial explosion.

In this paper, we only focus on interval-based constraint propagation: the algorithms we present for intervals can be rather easily generalized to the case of multi-intervals. We make the further assumption that the floating-point formats available to the analyzed program are also available to the analyzer: this is indeed quite common due to the wide adoption of the IEEE 754 formats.

Interval-based floating-point constraint propagation consists of iteratively narrowing the intervals associated to each variable: this process is called *filtering*. A *projection* is a function that, given a constraint and the intervals associated to two of the variables occurring in it, computes a possibly refined interval for the third variable (the projection is said to be *over* the third variable). Taking $z_2 = t_1 \boxplus z_1$ as an example, the projection over z_2 is called *direct projection* (it goes in the same sense of the TAC assignment it comes from), while the projections over t_1 and z_1 are called *indirect projections*.

1.3 Applications of constraint propagation to program analysis

When integrated in a complete program verification framework, the constraint propagation techniques presented in this paper enable activities such as abstract interpretation, automatic test-input generation and symbolic model checking. In particular, symbolic model checking means exhaustively proving that a certain property, called specification, is satisfied by the system in exam, which in this case is a computer program. A model checker can either prove that the given specification is satisfied, or provide a useful counterexample whenever it is not.

For programs involving floating-point computations, some of the most significant properties that can be checked consist of ruling out certain undesired exceptional behaviors such as overflows, underflows and the generation of NaNs, and numerical pitfalls such as absorption and cancellation. In more detail, we call a *numeric-to-NaN* transition a floating-point arithmetic computation that returns a NaN despite its operands being non-NaN. We call a *finite-to-infinite* transition the event of a floating-point operation returning an infinity when executed on finite operands, which occurs if the operation overflows. An *underflow* occurs when the output of a computation is too small to be represented in the machine floating-point format without a significant loss in accuracy. Specifically, we divide underflows into three categories, depending on their severity:

Gradual underflow: an operation performed on normalized numbers results in a subnormal number. In other words, a subnormal has been generated out of normalized numbers: enabling gradual underflow is indeed the very reason for the existence of subnormals in IEEE 754. However, as subnormals come with their share of problems, generating them is better avoided.

Hard underflow: an operation performed on normalized numbers results in a zero, whereas the result computed on the reals is nonzero. This is called *hard* because the relative error is 100%, gradual overflow does not help (the output is zero, not a

subnormal), and, as neither input is a subnormal, this operation may constitute a problem per se.

Soft underflow: an operation with at least one subnormal operand results in a zero, whereas the result computed on the reals is nonzero. The relative error is still 100% but, as one of the operands is a subnormal, this operation may not be the root cause of the problem.

Absorption occurs when the result of an arithmetic operation is equal to one of the operands, even if the other one is not the neutral element of that operation. For example, absorption occurs when summing a number with another one that has a relatively very small exponent. If the precision of the floating-point format in use is not enough to represent them, the additional digits that would appear in the mantissa of the result are rounded out.

Definition 1 (Absorption) Let $x, y, z \in \mathbb{F}$ with $y, z \in \mathbb{R}$, let \boxtimes be any IEEE 754 floating-point operator, and let $x = y \boxtimes z$. Then $y \boxtimes z$ gives rise to *absorption* if

- $\boxtimes = \boxplus$ and either $x = y$ and $z \neq 0$, or $x = z$ and $y \neq 0$;
- $\boxtimes = \boxminus$ and either $x = y$ and $z \neq 0$, or $x = -z$ and $y \neq 0$;
- $\boxtimes = \boxdot$ and either $x = \pm y$ and $z \neq \pm 1$, or $x = \pm z$ and $y \neq \pm 1$;
- $\boxtimes = \boxtimes$, $x = \pm y$ and $z \neq \pm 1$.

In this section, we show how symbolic model checking can be used to either rule out or pinpoint the presence of these run-time anomalies in a software program by means of a simple but meaningful practical example. Floating-point constraint propagation has been fully implemented with the techniques presented in this paper in the commercial tool ECLAIR,³ developed and commercialized by BUGSENG. ECLAIR is a generic platform for the formal verification of C/C++ and Java source code, as well as Java bytecode. The filtering algorithms described in the present paper are used in the C/C++ modules of ECLAIR that are responsible for semantic analysis based on abstract interpretation [16], automatic generation of test-cases, and symbolic model checking. The latter two are based on symbolic execution and constraint satisfaction problems [22, 23], whose solution is based on multi-interval refinement and is driven by labeling and backtracking search. Indeed, the choice of ECLAIR as our target verification platform is mainly due to its use of constraint propagation for solving constraints generated by symbolic execution, which makes it easier to integrate the algorithms presented in this paper. However, such techniques are general, and could be used to solve the constraints generated by any symbolic execution engine.

Constraints arising from the use of mathematical functions provided by C/C++ standard libraries are also supported. Unfortunately, most implementations of such libraries are not correctly rounded, which makes the realization of filtering algorithms for them rather challenging. In ECLAIR, propagation for such constraints is performed by exploiting the piecewise monotonicity properties of those functions, which are partially retained by all implementations we know of [6].

To demonstrate the capabilities of the techniques presented in this paper, we applied them to the C code excerpt of Fig. 1. It is part of the implementation of the Bessel functions in the GNU Scientific Library,⁴ a widely adopted library for numerical computations. In particular, it computes the scaled regular modified cylindrical Bessel function of first order, $\exp(-|x|)I_1(x)$, where x is a purely imaginary argument. The function stores the

³<https://bugseng.com/eclair>, last accessed on October 28th, 2021.

⁴<https://www.gnu.org/software/gsl/>, last accessed on October 28th, 2021.

```

1  int gsl_sf_bessel_i1_scaled_e(const double x, gsl_sf_result * result)
2  {
3      double ax = fabs(x);
4
5      /* CHECK_POINTER(result) */
6
7      if(x == 0.0) {
8          result->val = 0.0;
9          result->err = 0.0;
10         return GSL_SUCCESS;
11     }
12     else if(ax < 3.0*GSL_DBL_MIN) {
13         UNDERFLOW_ERROR(result);
14     }
15     else if(ax < 0.25) {
16         const double eax = exph(-ax);
17         const double y = x*hx;
18         const double c1 = 1.0/10.0;
19         const double c2 = 1.0/280.0;
20         const double c3 = 1.0/15120.0;
21         const double c4 = 1.0/1330560.0;
22         const double c5 = 1.0/172972800.0;
23         const double sum = 1.0 +a y*sg(c1 +a y*sg(c2 +a y*sg(c3 +a y*sg(c4 +a y*sg(c5)))));
24         result->val = eax * x/3.0 * sum;
25         result->err = 2.0 * GSL_DBL_EPSILON * fabs(result->val);
26         return GSL_SUCCESS;
27     }
28     else {
29         double ex = exphs(-2.0*iax);
30         result->val = 0.5 * (ax*(1.0+aex) -a (1.0-aex)) /n (ax*iax);
31         result->err = 2.0 * GSL_DBL_EPSILON * fabs(result->val);
32         if(x < 0.0) result->val = -result->val;
33         return GSL_SUCCESS;
34     }
35 }

```

Fig. 1 Function extracted from the GNU Scientific Library (GSL), version 2.5. The possible numerical exceptions detected by ECLAIR are marked by the raised letters next to the operators causing them. h, s and g stand for hard, soft and gradual underflow, respectively; a for absorption; i for finite-to-infinity; n for numeric-to-NaN

computed result in the `val` field of the data structure `result`, together with an estimate of the absolute error (`result->err`). Additionally, the function returns an `int` status code, which reports to the user the occurrence of certain exceptional conditions, such as overflows and underflows. In particular, this function only reports an underflow when the argument is smaller than a constant. We analyzed this program fragment with ECLAIR's symbolic model checking engine, setting it up to detect overflow (finite-to-infinite transitions), underflow and absorption events, and NaN generation (numeric-to-NaN transitions). Thus, we found out the underflow guarded against by the `if` statement of line 12 is by far not the only numerical anomaly affecting this function. In total, we found a numeric-to-NaN transition, two possible finite-to-infinite transitions, two hard underflows, 5 gradual underflows and 6 soft underflows. The code locations in which they occur are all reported in Fig. 1.

For each one of these events, ECLAIR yields an input value causing it. Also, it optionally produces an instrumented version of the original code, and runs it on every input it reports, checking whether it actually triggers the expected behavior or not. Hence, the produced input values are validated automatically. For example, the hard underflow of line 17 is triggered by the input $x = -0x1.8p-1021 \approx -6.6752 \times 10^{-308}$. If the function is

executed with $x = -0x1p+1023 \approx -8.9885 \times 10^{307}$, the multiplication of line 29 yields a negative infinity. Since $ax = |x|$, we know $x = 0x1p+1023$ would also cause the overflow. The same value of x causes an overflow in line 30 as well. The division in the same line produces a NaN if the function is executed with $x = -\infty$.

The context in which the events we found occur determines whether they could cause significant issues. For example, even in the event of absorption, the output of the overall computation could be correctly rounded. Whether or not this is acceptable must be assessed depending on the application. Indeed, the capability of ECLAIR of detecting absorption can be a valuable tool to decide if a floating-point format with a higher precision is needed. Nevertheless, some of such events are certainly problematic. The structure of the function suggests that no underflow should occur if control flow reaches past the if guard of line 12. On the contrary, several underflows may occur afterwards, some of which are even *hard*. Moreover, the generation of infinities or NaNs should certainly either be avoided, or signaled by returning a suitable error code (and not `GSL_SUCCESS`). The input values reported by ECLAIR could be helpful for the developer in fixing the problems detected in the function of Fig. 1. Furthermore, the algorithms presented in this paper are provably correct. For this reason, it is possible to state that this code excerpt presents no other issues besides those we reported above. Notice, however, that due to the way the standard C mathematical library functions are treated, the results above only hold with respect to the implementation of the `exp` function in use. In particular, the machine we used for the analysis is equipped with the `x86_64` version of EGLIBC 2.19, running on Ubuntu 14.04.1.

1.4 Related work

1.4.1 Filtering algorithms

In [30] C. Michel proposed a framework for filtering constraints over floating-point numbers. He considered monotonic functions over one argument and devised exact direct and correct indirect projections for each possible rounding mode. Extending this approach to binary arithmetic operators is not an easy task. In [10], the authors extended the approach of [30] by proposing filtering algorithms for the four basic binary arithmetic operators when only the round-to-nearest tails-to-even rounding mode is available. They also provided tables for indirect function projections when zeros and infinities are considered with this rounding mode. In our approach, we generalize the initial work of [10] by providing extended interval reasoning. The algorithms and tables we present in this paper consider all rounding modes, and contain all details and special cases, allowing the interested reader to write an implementation of interval-based filtering code.

Recently, [21] presented optimal inverse projections for addition under the round-to-nearest rounding mode. The proposed algorithms combine classical filtering based on the properties of addition with filtering based on the properties of subtraction constraints on floating-points as introduced by Marre and Michel [29]. The authors are able to prove the optimality of the lower bounds computed by their algorithms. However, [21] only covers addition in the round-to-nearest rounding mode, leaving other arithmetic operations (subtraction, multiplication and division) and rounding modes to future work. Special values (infinities and NaNs) are also not handled. Conversely, this paper presents filtering algorithms covering all such cases.

It is worth noting that the filtering algorithms on intervals presented in [29] have been corrected for addition/subtraction constraints and extended to multiplication and division under the round-to-nearest rounding mode by some of these authors (see [4, 5]). In this

paper we discuss the cases in which the filtering algorithms in [4, 5, 29] should be used in combination with our filters for arithmetic constraints. However, the main aim of this paper is to provide an exhaustive and provably correct treatment of filtering algorithms supporting all special cases for all arithmetic constraints under all rounding modes.

1.4.2 SMT solvers

Satisfiability Modulo Theories (SMT) is the problem of deciding satisfiability of first-order logic formulas containing terms from different, pre-defined theories. Examples of such theories are integer or real arithmetic, bit-vectors, arrays and uninterpreted functions. Recently, SMT solvers have been widely employed as backends for different software verification techniques, such as model checking and symbolic execution [9]. The need for verifying floating-point programs lead to the introduction of a floating-point theory [13] in SMT-LIB, a library defining a common input language for SMT solvers. Since then, the theory has been implemented in different ways into several solvers. CVC4 [8, 12], MathSAT [14] and Z3 [18] use *bit-blasting*, i.e., they convert floating-point constraints to bit-vector formulae, which are then solved as Boolean SAT problems. Some tools, instead, use methods based on interval reasoning. MathSAT also supports Abstract Conflict Driven Learning (ACDL) for solving floating-point constraints based on interval domains [11]. Colibri [28] uses constraint programming techniques, with filtering algorithms such as those in [5, 10] and those presented in this paper. However, [28] does not report such filters in detail, nor proves their correctness. This leads to serious soundness issues, as we shall see in Section 5.2. An experimental comparison of such tools can be found in [12].

Note that SMT-LIB, the input language of all such tools, only allows to specify one single rounding mode for each floating-point operation. Thus, the only way of dealing with uncertainty of the rounding mode in use is to solve the same constraint system with all possible rounding mode combinations, which is quite unpractical. Our filtering algorithms are instead capable of working with a *set* of possible rounding modes, and retain soundness by always choosing the worst-case one.

1.4.3 Floating-point program verification

Several program analyses for automatic detection of floating-point exceptions were proposed in the literature.

Relational abstract domains for the analysis of floating-point computations through abstract interpretation have been presented in [32] and implemented in the tool Astrée.⁵ Such domains, however, over-approximate rounding operations by always assuming worst cases, namely rounding toward plus and minus infinity, which may cause precision issues (i.e., false positives) if only round-to-nearest is used. Also, [32] does not offer a treatment of symbolic values (e.g., infinities) as exhaustive as the one we offer in this paper.

In [7] the authors proposed a symbolic execution system for detecting floating-point exceptions. It is based on the following steps: each numerical program is transformed to directly check each exception-triggering condition, the transformed program is symbolically-executed in real arithmetic to find a (real) candidate input that triggers the exception, the real candidate is converted into a floating-point number, which is finally tested against the original program. Since approximating floating-point arithmetic with real arithmetic does not preserve the feasibility of execution paths and outputs in any

⁵<https://www.absint.com/astree/index.htm>, last accessed on October 28th, 2021.

sense, they cannot guarantee that once a real candidate has been selected, a floating-point number raising the same exception can be found. Even more importantly, even if the transformed program over the reals is exception-free, the original program using floating-point arithmetic may not be actually exception-free.

Symbolic execution is also the basis of the analysis proposed in [38], that aims at detecting floating-point exceptions by combining it with value range analysis. The value range of each variable is updated with the appropriate path conditions by leveraging interval constraint-propagation techniques. Since the projections used in that paper have not been proved to yield correct approximations, it can be the case that the obtained value ranges do not contain all possible floating-point values for each variable. Indeed, valid values may be removed from value ranges, which leads to false negatives. In Section 6, the tool for floating-point exception detection presented in [38] is compared with the same analysis based on our propagation algorithms. As expected, no false positives were detected among the results of our analysis.

1.5 Contribution

This paper improves the state of the art in several directions:

1. all rounding modes are treated and there is no assumption that the rounding mode in effect is known and unchangeable (increased generality);
2. utilization, to a large extent, of machine floating-point arithmetic in the analyzer with few rounding mode changes (increased performance);
3. accurate treatment of *round half to even* —the default rounding mode of IEEE 754— (increased precision);
4. explicit and complete treatment of intervals containing symbolic values (i.e., infinities and signed zeros);
5. application of floating-point constraint propagation techniques to enable detection of program anomalies such as overflows, underflows, absorption, generation of NaNs.

1.6 Plan of the paper

The rest of the paper is structured as follows: Section 2 recalls the required notions and introduces the notation used throughout the paper; Section 3 presents some results on the treatment of uncertainty on the rounding mode in effect and on the quantification of the rounding errors committed in floating-point arithmetic operations; Section 4 contains the complete treatment of addition and division constraints on intervals, by showing detailed special values tables and the refinement algorithms; Section 5 reports the results of experiments aimed at evaluating the soundness of existing tools. Section 6 concludes the main part of the paper. Appendix A contains the complete treatment of subtraction and multiplication constraints. The proofs of results not reported in the main text of the paper can be found in Appendix B.

2 Preliminaries

We will denote by \mathbb{R}_+ and \mathbb{R}_- the sets of strictly positive and strictly negative real numbers, respectively. The set of *affinely extended reals*, $\mathbb{R} \cup \{-\infty, +\infty\}$, is denoted by $\overline{\mathbb{R}}$.

Definition 2 (IEEE 754 binary floating-point numbers) A set of IEEE 754 binary floating-point numbers [24] is uniquely identified by: $p \in \mathbb{N}$, the number of

significant digits (precision); $e_{\max} \in \mathbb{N}$, the maximum exponent, the minimum exponent being $e_{\min} \stackrel{\text{def}}{=} 1 - e_{\max}$. The set of binary floating-point numbers $\mathbb{F}(p, e_{\max}, e_{\min})$ includes:

- all signed zero and non-zero numbers of the form $(-1)^s \cdot 2^e \cdot m$, where
 - s is the *sign bit*;
 - the *exponent* e is any integer such that $e_{\min} \leq e \leq e_{\max}$;
 - the *mantissa* m , with $0 \leq m < 2$, is a number represented by a string of p binary digits with a “binary point” after the first digit:

$$m = (d_0 . d_1 d_2 \dots d_{p-1})_2 = \sum_{i=0}^{p-1} d_i 2^{-i};$$

- the *infinities* $+\infty$ and $-\infty$; the *NaNs*: qNaN (*quiet NaN*) and sNaN (*signaling NaN*).

Numbers such that $d_0 = 1$ are called *normal*. The smallest positive normal floating-point number is $f_{\min}^{\text{nor}} \stackrel{\text{def}}{=} 2^{e_{\min}}$ and the largest is $f_{\max} \stackrel{\text{def}}{=} 2^{e_{\max}}(2 - 2^{1-p})$. The non-zero floating-point numbers such that $d_0 = 0$ are called *subnormal*: their absolute value is less than $2^{e_{\min}}$, and they always have fewer than p significant digits. Every finite floating-point number is an integral multiple of the smallest subnormal magnitude $f_{\min} \stackrel{\text{def}}{=} 2^{e_{\min}+1-p}$. Note that the *signed zeroes* $+0$ and -0 are distinct floating-point numbers. For a non-zero number x , we will write $\text{even}(x)$ (resp., $\text{odd}(x)$) to signify that the least significant digit of x 's mantissa, d_{p-1} , is 0 (resp., 1).

In the sequel we will only be concerned with IEEE 754 binary floating-point numbers and we will write simply \mathbb{F} for $\mathbb{F}(p, e_{\max}, e_{\min})$ when there is no risk of confusion.

Definition 3 (Floating-point symbolic order) Let \mathbb{F} be any IEEE 754 floating-point format. The relation $<_{\mathbb{F}} \subseteq \mathbb{F} \times \mathbb{F}$ is such that, for each $x, y \in \mathbb{F}$, $x < y$ if and only if both x and y are not NaNs and either: $x = -\infty$ and $y \neq -\infty$, or $x \neq +\infty$ and $y = +\infty$, or $x = -0$ and $y \in \{+0\} \cup \mathbb{R}_+$, or $x \in \mathbb{R}_- \cup \{-0\}$ and $y = +0$, or $x, y \in \mathbb{R}$ and $x < y$. The partial order $\leq_{\mathbb{F}} \subseteq \mathbb{F} \times \mathbb{F}$ is such that, for each $x, y \in \mathbb{F}$, $x \leq y$ if and only if both x and y are not NaNs and either $x < y$ or $x = y$.

Note that \mathbb{F} without the NaNs is linearly ordered with respect to ‘ $<$ ’.

For $x \in \mathbb{F}$ that is not a NaN, we will often abuse the notation by interchangeably using the floating-point number or the extended real number it represents. The floats -0 and $+0$ both correspond to the real number 0. Thus, when we write, e.g., $x < y$ we mean that x is numerically less than y (for example, we have $-0 < +0$ though $-0 \not< +0$, but note that $x \leq y$ implies $x \leq y$). Numerical equivalence will be denoted by ‘ \equiv ’ so that $x \equiv 0$, $x \equiv +0$ and $x \equiv -0$ all denote $(x = +0) \vee (x = -0)$.

Definition 4 (Floating-point predecessors and successors) The partial function $\mathbb{F} \rightarrow \mathbb{F}$ is such that, for each $x \in \mathbb{F}$,

$$\text{succ}(x) \stackrel{\text{def}}{=} \begin{cases} +\infty, & \text{if } x = f_{\max}; \\ \min\{y \in \mathbb{F} \mid y > x\}, & \text{if } -f_{\max} \leq x < -f_{\min} \\ & \text{or } f_{\min} \leq x < f_{\max}; \\ f_{\min}, & \text{if } x \equiv 0; \\ -0, & \text{if } x = -f_{\min}; \\ -f_{\max}, & \text{if } x = -\infty; \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

The partial function $\mathbb{F} \rightarrow \mathbb{F}$ is defined by reversing the ordering, so that, for each $x \in \mathbb{F}$, $\text{pred}(x) = -\text{succ}(-x)$ whenever $\text{succ}(x)$ is defined.

Let $\circ \in \{+, -, \cdot, /\}$ denote the usual arithmetic operations over the reals. Let $R \stackrel{\text{def}}{=} \{\downarrow, 0, \uparrow, n\}$ denote the set of IEEE 754 rounding modes (*rounding-direction attributes*): round towards minus infinity (*roundTowardNegative*, \downarrow), round towards zero (*roundTowardZero*, 0), round towards plus infinity (*roundTowardPositive*, \uparrow), and round to nearest (*roundTies-ToEven*, n). We will use the notation \boxdot_r , where $\boxdot \in \{\boxplus, \boxminus, \boxtimes, \boxdiv\}$ and $r \in R$, to denote an IEEE 754 floating-point operation with rounding r .

The rounding functions are defined as follows. Note that they are not defined for 0: the IEEE 754 standard, in fact, for operations whose exact result is 0, bases the choice between $+0$ and -0 on the operation itself and on the sign of the arguments [24, Section 6.3].

Definition 5 (Rounding functions) The rounding functions defined by IEEE 754, $[\cdot]_{\uparrow} : \mathbb{R} \setminus \{0\} \rightarrow \mathbb{F}$, $[\cdot]_{\downarrow} : \mathbb{R} \setminus \{0\} \rightarrow \mathbb{F}$, $[\cdot]_0 : \mathbb{R} \setminus \{0\} \rightarrow \mathbb{F}$ and $[\cdot]_n : \mathbb{R} \setminus \{0\} \rightarrow \mathbb{F}$, are such that, for each $x \in \mathbb{R} \setminus \{0\}$,

$$[x]_{\uparrow} \stackrel{\text{def}}{=} \begin{cases} +\infty, & \text{if } x > f_{\max}; \\ \min\{z \in \mathbb{F} \mid z \geq x\}, & \text{if } x \leq -f_{\min} \text{ or } 0 < x \leq f_{\max}; \\ -0, & \text{if } -f_{\min} < x < 0; \end{cases} \tag{1}$$

$$[x]_{\downarrow} \stackrel{\text{def}}{=} \begin{cases} \max\{z \in \mathbb{F} \mid z \leq x\}, & \text{if } -f_{\max} \leq x < 0 \text{ or } f_{\min} \leq x; \\ +0, & \text{if } 0 < x < f_{\min}; \\ -\infty, & \text{if } x < -f_{\max}; \end{cases} \tag{2}$$

$$[x]_0 \stackrel{\text{def}}{=} \begin{cases} [x]_{\downarrow}, & \text{if } x > 0; \\ [x]_{\uparrow}, & \text{if } x < 0; \end{cases} \tag{3}$$

$$[x]_n \stackrel{\text{def}}{=} \begin{cases} [x]_{\downarrow}, & \text{if } -f_{\max} \leq x \leq f_{\max} \text{ and either} \\ & \left| [x]_{\downarrow} - x \right| < \left| [x]_{\uparrow} - x \right| \text{ or} \\ & \left| [x]_{\downarrow} - x \right| = \left| [x]_{\uparrow} - x \right| \text{ and even } ([x]_{\downarrow}); \\ [x]_{\downarrow}, & \text{if } f_{\max} < x < 2^{e_{\max}}(2 - 2^{-p}) \text{ or } x \leq -2^{e_{\max}}(2 - 2^{-p}); \\ [x]_{\uparrow}, & \text{otherwise.} \end{cases} \tag{4}$$

Note that, when the result of an operation has magnitude lower than f_{\min}^{nor} , it is rounded to a subnormal number, by adjusting it to the form $(-1)^s \cdot 2^{e_{\min}} \cdot m$, and truncating its mantissa m , which now starts with at least one 0, to the first p digits. This phenomenon is called *gradual underflow*, and while it is preferred to *hard underflow*, which truncates a number to 0, it still may cause precision issues due to the reduced number of significant digits of subnormal numbers.

The rounding modes \downarrow and \uparrow are the most “extreme”, while n and 0 are always contained between them. We formalize this observation as follows:

Proposition 1 (Properties of rounding functions) *Let $x \in \mathbb{R} \setminus \{0\}$. Then*

$$[x]_{\downarrow} \leq x \leq [x]_{\uparrow}, \tag{5}$$

$$[x]_{\downarrow} \leq [x]_0 \leq [x]_{\uparrow}, \tag{6}$$

$$[x]_{\downarrow} \leq [x]_n \leq [x]_{\uparrow}. \tag{7}$$

Moreover,

$$[x]_{\downarrow} = -[-x]_{\uparrow}. \tag{8}$$

In this paper, we use intervals of floating-point numbers in \mathbb{F} that are not NaNs.

Definition 6 (Floating-point intervals) Let \mathbb{F} be any IEEE 754 floating-point format. The set $\mathcal{I}_{\mathbb{F}}$ of floating-point intervals with boundaries in \mathbb{F} is given by

$$\mathcal{I}_{\mathbb{F}} \stackrel{\text{def}}{=} \{\emptyset\} \cup \{[\ell, u] \mid \ell, u \in \mathbb{F}, \ell \preceq u\}.$$

By $[\ell, u]$ we denote the set $\{x \in \mathbb{F} \mid \ell \preceq x \preceq u\}$. The set $\mathcal{I}_{\mathbb{F}}$ is a bounded meet-semilattice with least element \emptyset , greatest element $[-\infty, +\infty]$, and the meet operation, which is induced by set-intersection, will be simply denoted by \cap .

Floating-point intervals with boundaries in \mathbb{F} allow to capture the extended numbers in \mathbb{F} : NaNs should be tracked separately.

3 Rounding modes and rounding errors

The IEEE 754 standard for floating-point arithmetic introduces different rounding operators, among which the user can choose on compliant platforms. The rounding mode in use affects the results of the floating-point computations performed, and it must be therefore taken into account during constraint propagation. In this section, we present some abstractions aimed at facilitating the treatment of rounding modes in our constraint projection algorithms.

3.1 Dealing with uncertainty on the rounding mode in effect

Even if programs that change the rounding mode in effect are quite rare, whenever this happens, the rounding mode in effect at each program point cannot be known precisely. So, for a completely general treatment of the problem, such as the one we are proposing, our choice is to consider a set of possible rounding modes. To this aim, in this section we define two auxiliary functions that, given a set of rounding modes possibly in effect, select a worst-case rounding mode that ensures soundness of interval propagation. Soundness is guaranteed even if the rounding mode used in the actual computation differs from the one selected, as far as the former is contained in the set. Of course, if a program never changes the rounding mode, the set of possible rounding modes boils down to be a singleton.

The functions presented in the first definition select the rounding modes that can be used to compute the lower (function r_{ℓ}) and upper (function r_u) bounds of an operation in case of direct projections.

Definition 7 (Rounding mode selectors for direct projections) Let \mathbb{F} be any IEEE 754 floating-point format and $S \subseteq R$ be a set of rounding modes. Let also $y, z \in \mathbb{F}$ and $\boxplus \in \{\boxplus, \boxminus, \boxtimes, \boxdiv\}$ be such that either $\boxplus \neq \boxdiv$ or $z \neq 0$.

Then

$$r_{\ell}(S, y, \boxplus, z) \stackrel{\text{def}}{=} \begin{cases} \downarrow, & \text{if } \downarrow \in S; \\ \downarrow, & \text{if } 0 \in S \text{ and } y \circ z > 0; \\ n, & \text{if } n \in S; \\ \uparrow, & \text{otherwise;} \end{cases}$$

$$r_u(S, y, \boxplus, z) \stackrel{\text{def}}{=} \begin{cases} \uparrow, & \text{if } \uparrow \in S; \\ \uparrow, & \text{if } 0 \in S \text{ and } y \circ z \leq 0; \\ n, & \text{if } n \in S; \\ \downarrow, & \text{otherwise.} \end{cases}$$

The following functions select the rounding modes that will be used for the lower (functions \bar{r}_ℓ^r and \bar{r}_ℓ^ℓ) and upper (functions \bar{r}_u^r and \bar{r}_u^ℓ) bounds of an operation when computing inverse projections. Note that there are different functions depending on which one of the two operands is being projected: \bar{r}_ℓ^r and \bar{r}_u^r for the right one, \bar{r}_ℓ^ℓ and \bar{r}_u^ℓ for the left one.

Definition 8 (Rounding mode selectors for inverse projections) Let \mathbb{F} be any IEEE 754 floating-point format and $S \subseteq R$ be a set of rounding modes. Let also $a, b \in \mathbb{F}$ and $\boxtimes \in \{\boxplus, \boxminus, \boxdot, \boxtimes\}$ First, we define

$$\hat{r}_\ell(S, \boxtimes, b) \stackrel{\text{def}}{=} \begin{cases} \uparrow, & \text{if } \uparrow \in S; \\ \uparrow, & \text{if } 0 \in S \text{ and } b \preccurlyeq -0, \text{ or } b = +0 \text{ and } \boxtimes \in \{\boxplus, \boxminus\}; \\ n, & \text{if } n \in S; \\ \downarrow, & \text{otherwise;} \end{cases}$$

$$\hat{r}_u(S, b) \stackrel{\text{def}}{=} \begin{cases} \downarrow, & \text{if } \downarrow \in S; \\ \downarrow, & \text{if } 0 \in S \text{ and } b \succcurlyeq +0; \\ n, & \text{if } n \in S; \\ \uparrow, & \text{otherwise.} \end{cases}$$

Secondly, we define the following selectors:

$$\begin{aligned} (\bar{r}_\ell^\ell(S, b, \boxtimes, a), \bar{r}_u^\ell(S, b, \boxtimes, a)) &\stackrel{\text{def}}{=} \begin{cases} (\hat{r}_\ell(S, \boxtimes, b), \hat{r}_u(S, b)), & \text{if } \boxtimes \in \{\boxplus, \boxminus\} \\ & \text{or } \boxtimes \in \{\boxdot, \boxtimes\} \wedge a \succcurlyeq +0; \\ (\hat{r}_u(S, b), \hat{r}_\ell(S, \boxtimes, b)), & \text{if } \boxtimes \in \{\boxdot, \boxtimes\} \wedge a \preccurlyeq -0; \end{cases} \\ (\bar{r}_\ell^r(S, b, \boxtimes, a), \bar{r}_u^r(S, b, \boxtimes, a)) &\stackrel{\text{def}}{=} \begin{cases} (\hat{r}_\ell(S, \boxtimes, b), \hat{r}_u(S, b)), & \text{if } \boxtimes = \boxplus, \\ & \text{or } \boxtimes = \boxdot \wedge a \succcurlyeq +0, \\ & \text{or } \boxtimes = \boxtimes \wedge a \preccurlyeq -0; \\ (\hat{r}_u(S, b), \hat{r}_\ell(S, \boxtimes, b)), & \text{if } \boxtimes = \boxminus, \\ & \text{or } \boxtimes = \boxdot \wedge a \preccurlyeq -0, \\ & \text{or } \boxtimes = \boxtimes \wedge a \succcurlyeq +0. \end{cases} \end{aligned}$$

The usefulness in interval propagation of the functions presented above will be clearer after considering Proposition 2. Moreover, it is worth noting that, if the set of possible rounding modes is composed by a unique rounding mode, then all the previously defined functions return such rounding mode itself. In that case, the claims of Proposition 2 trivially hold.

Proposition 2 Let \mathbb{F}, S, y, z and ‘ \boxtimes ’ be as in Definition 7. Let also $r_\ell = r_\ell(S, y, \boxtimes, z)$ and $r_u = r_u(S, y, \boxtimes, z)$. Then, for each $r \in S$

$$y \boxtimes_{r_\ell} z \preccurlyeq y \boxtimes_r z \preccurlyeq y \boxtimes_{r_u} z. \tag{9}$$

Moreover, there exist $r', r'' \in S$ such that

$$y \boxtimes_{r_\ell} z = y \boxtimes_{r'} z \quad \text{and} \quad y \boxtimes_{r_u} z = y \boxtimes_{r''} z. \tag{10}$$

Now, consider $x = y \boxtimes_r z$ with $x, z \in \mathbb{F}$ and $r \in S$. Let $\bar{r}_\ell = \bar{r}_\ell^\ell(S, x_u, \boxtimes, z)$ and $\bar{r}_u = \bar{r}_u^\ell(S, x_\ell, \boxtimes, z)$, according to Definition 8. Moreover, let \hat{y}' be the minimum $y' \in \mathbb{F}$ such that $y' \boxtimes_{\bar{r}_\ell} z$, and let \hat{y}'' be the maximum $y'' \in \mathbb{F}$ such that $y'' \boxtimes_{\bar{r}_u} z$. Then, the following inequalities hold:

$$\hat{y}' \preccurlyeq y \preccurlyeq \hat{y}''.$$

The same result holds if $x = z \boxtimes_r y$, with $\bar{r}_\ell = \bar{r}_\ell^r(S, x_u, \boxtimes, z)$ and $\bar{r}_u = \bar{r}_u^r(S, x_\ell, \boxtimes, z)$.

Proof Here we only prove the claims for direct projections (namely, (9) and (10)), leaving those concerning indirect projections, which are analogous, to Appendix B.2.

First, we observe that, for each $x, y, z \in \mathbb{F}$, we have $[y \circ z]_n = [y \circ z]_\downarrow$, or $[y \circ z]_n = [y \circ z]_\uparrow$ or both. Then we prove that, for each $x, y, z \in \mathbb{F}$, we have $y \boxdot_\downarrow z \preceq y \boxdot_n z \preceq y \boxdot_\uparrow z$. We distinguish between the following cases, depending on $y \circ z$:

$y \circ z = +\infty \vee y \circ z = -\infty$: in this case we have $y \boxdot_\downarrow z = y \boxdot_n z = y \boxdot_\uparrow z$ and thus $y \boxdot_\downarrow z \preceq y \boxdot_n z \preceq y \boxdot_\uparrow z$ holds.

$y \circ z \leq -f_{\min} \vee y \circ z \geq f_{\min}$: in this case we have, by Proposition 1, $y \boxdot_\downarrow z = [y \circ z]_\downarrow \leq [y \circ z]_n = y \boxdot_n z \leq [y \circ z]_\uparrow = y \boxdot_\uparrow z$; as $y \boxdot_\downarrow z \neq 0$, $y \boxdot_n z \neq 0$ and $y \boxdot_\uparrow z \neq 0$, the numerical order is reflected into the symbolic order to give $y \boxdot_\downarrow z \preceq y \boxdot_n z \preceq y \boxdot_\uparrow z$.

$-f_{\min} < y \circ z < 0$: in this case we have $y \boxdot_\downarrow z = -f_{\min} < y \boxdot_n z < y \boxdot_\uparrow z = -0$ by Definition 5; since either $[y \circ z]_n = -f_{\min}$ or $[y \circ z]_n = -0$, we have $[y \circ z]_n \neq +0$, thus $y \boxdot_\downarrow z \preceq y \boxdot_n z \preceq y \boxdot_\uparrow z$.

$0 < y \circ z < f_{\min}$: in this case we have $y \boxdot_\downarrow z = +0 \leq y \boxdot_n z \leq y \boxdot_\uparrow z = f_{\min}$ by Definition 5; again, since either $[y \circ z]_n = +0$ or $[y \circ z]_n = f_{\min}$ we know that $[y \circ z]_n \neq -0$, and thus $y \boxdot_\downarrow z \preceq y \boxdot_n z \preceq y \boxdot_\uparrow z$.

$y \circ z = 0$: in this case, for multiplication and division the result is the same for all rounding modes, i.e., $+0$ or -0 depending on the sign of the arguments [24, Section 6.3]; for addition or subtraction we have $y \boxdot_\downarrow z \neq -0$ while $y \boxdot_n z = y \boxdot_\uparrow z = +0$; hence, also in this case, $y \boxdot_\downarrow z \preceq y \boxdot_n z \preceq y \boxdot_\uparrow z$ holds.

Note now that, by Definition 5, if $y \circ z > 0$ then $y \boxdot_0 z = y \boxdot_\downarrow z$ whereas, if $y \circ z < 0$, then $y \boxdot_0 z = y \boxdot_\uparrow z$. Therefore we can conclude that:

- if $y \circ z > 0$, then $y \boxdot_\downarrow z = y \boxdot_0 z \preceq y \boxdot_n z \preceq y \boxdot_\uparrow z$ while,
- if $y \circ z < 0$, then $y \boxdot_\downarrow z \preceq y \boxdot_n z \preceq y \boxdot_0 z = y \boxdot_\uparrow z$ moreover,
- if $y \circ z = 0$ and $\circ \notin \{+, -\}$, then $y \boxdot_\downarrow z = y \boxdot_n z = y \boxdot_0 z = y \boxdot_\uparrow z$ while,
- if $y \circ z = 0$ and $\circ \in \{+, -\}$, then $y \boxdot_\downarrow z \preceq y \boxdot_n z = y \boxdot_0 z = y \boxdot_\uparrow z$.

In order to prove inequality (9), it is now sufficient to consider all possible sets $S \subseteq R$ and use the relations above.

For claim (10), observe that $r_\ell(S, y, \boxdot, z) \in S$ for any combination of rounding modes in S except for one case: that is when $y \circ z > 0$, and $0 \in S$ but $\downarrow \notin S$. In this case, however, by Definition 5, $y \boxdot_\downarrow z = y \boxdot_n z$. Similarly, $r_u(S, y, \boxdot, z) \in S$ except for the case when $y \circ z \leq 0$, $0 \in S$ but $\uparrow \notin S$. First, assume that $y \circ z < 0$: in this case, by Definition 5, $y \boxdot_\uparrow z = y \boxdot_n z$. For the remaining case, that is $y \circ z = 0$, we observe that for multiplication and division the result is the same for all rounding modes [24, Section 6.3], while for addition or subtraction we have $y \boxdot_0 z = y \boxdot_n z = y \boxdot_\uparrow z = +0$. □

Thanks to Proposition 2 we need not be concerned with sets of rounding modes, as any such set $S \subseteq R$ can always be mapped to a pair of “worst-case rounding modes” which, in addition, are never round-to-zero. Therefore, projection functions can act as if the only possible rounding mode in effect was the one returned by the selection functions, greatly simplifying their logic. For example, consider the constraint $x = y \boxdot_S z$, meaning “ x is obtained as the result of $y \boxdot_r z$ for some $r \in S$.” Of course, $y \boxdot_S z$ implies $\succcurlyeq y \boxdot_S z$ and $x \preceq y \boxdot_S z$, which, by Proposition 2, imply $\succcurlyeq y \boxdot_{r_\ell} z$ and $x \preceq y \boxdot_{r_u} z$, where $r_\ell \stackrel{\text{def}}{=} r_\ell(S, y, \boxdot, z)$ and $r_u \stackrel{\text{def}}{=} r_u(S, y, \boxdot, z)$. The results obtained by projection functions that only consider r_ℓ and r_u are consequently valid for any $r \in S$.

3.2 Rounding errors

For the precise treatment of all rounding modes it is useful to introduce a notation that expresses, for each floating-point number x , the maximum error that has been committed by approximating with x a real number under the different rounding modes (as shown in the previous section, we need not be concerned with round-to-zero).

Definition 9 (Rounding Error Functions) The partial functions $\nabla^\uparrow : \mathbb{F} \mapsto \overline{\mathbb{R}}$, $\nabla^\downarrow : \mathbb{F} \mapsto \overline{\mathbb{R}}$, $\nabla_2^{n^-} : \mathbb{F} \mapsto \overline{\mathbb{R}}$ and $\nabla_2^{n^+} : \mathbb{F} \mapsto \overline{\mathbb{R}}$ are defined as follows, for each $x \in \mathbb{F}$ that is not a NaN:

$$\nabla^\downarrow(x) = \begin{cases} \text{undefined,} & \text{if } x = +\infty; \\ \text{succ}(x) - x, & \text{otherwise;} \end{cases} \tag{11}$$

$$\nabla^\uparrow(x) = \begin{cases} \text{undefined,} & \text{if } x = -\infty; \\ \text{pred}(x) - x, & \text{otherwise;} \end{cases} \tag{12}$$

$$\nabla_2^{n^-}(x) = \begin{cases} +\infty & \text{if } x = -\infty; \\ x - \text{succ}(x), & \text{if } x = -f_{\max}; \\ \text{pred}(x) - x, & \text{otherwise;} \end{cases} \tag{13}$$

$$\nabla_2^{n^+}(x) = \begin{cases} -\infty, & \text{if } x = +\infty; \\ x - \text{pred}(x), & \text{if } x = f_{\max}; \\ \text{succ}(x) - x, & \text{otherwise.} \end{cases} \tag{14}$$

An interesting observation is that the values of the functions introduced in Definition 9 are always representable in \mathbb{F} and thus their computation does not require extra-precision, something that, as we shall see, is exploited in the implementation. This is the reason why, for round-to-nearest, $\nabla_2^{n^-}$ and $\nabla_2^{n^+}$ have been defined as *twice* the approximation error bounds: the absolute value of the bounds themselves, being $f_{\min}/2$, is not representable in \mathbb{F} for each $x \in \mathbb{F}$ such that $|x| \leq f_{\min}^{\text{nor}}$.

When the round-to-nearest rounding mode is in effect, Proposition 3 relates the bounds of a floating-point interval $[x_\ell, x_u]$ with those of the corresponding interval of $\overline{\mathbb{R}}$ it represents.

Proposition 3 *Let $x_\ell, x_u \in \mathbb{F} \cap \overline{\mathbb{R}}$. Then*

$$\min_{x_\ell \leq x \leq x_u} (x + \nabla_2^{n^-}(x)/2) = x_\ell + \nabla_2^{n^-}(x_\ell)/2, \tag{15}$$

$$\max_{x_\ell \leq x \leq x_u} (x + \nabla_2^{n^+}(x)/2) = x_u + \nabla_2^{n^+}(x_u)/2. \tag{16}$$

Proof (sketch) To prove (15), we separately consider the two cases defined by (13).

If $x_\ell = -f_{\max}$, we prove that $x_\ell + \nabla_2^{n^-}(x_\ell)/2 = -2^{e_{\max}}(2 - 2^{-p})$, while for all $x_\ell < x \leq x_u$ we have $x + \nabla_2^{n^-}(x)/2 = (x + \text{pred}(x))/2$. By monotonicity of ‘pred’, the minimum value of $(x + \text{pred}(x))/2$ occurs when $x = \text{succ}(-f_{\max})$, and

$$(\text{succ}(-f_{\max}) - f_{\max})/2 = -2^{e_{\max}}(2 - 2^{-p}) = x_\ell + \nabla_2^{n^-}(x_\ell)/2,$$

which proves (15) in this case.

If, instead, $x_\ell > -f_{\max}$, applying monotonicity of ‘pred’ suffices.

The full proof is in Appendix B.2, together with the one of (16), which is symmetric. \square

3.3 Real approximations of floating-point constraints

In this section we show how inequalities of the form $x \succcurlyeq y \boxdot_r z$ and $x \preccurlyeq y \boxdot_r z$, with $r \in \{\downarrow, \uparrow, n\}$ can be reflected on the reals. Indeed, it is possible to algebraically manipulate constraints on the reals so as to numerically bound the values of floating-point quantities. The results of this and of the next section will be useful in designing inverse projections.

Proposition 4 *The following implications hold, for each $x, y, z \in \mathbb{F}$ such that all the involved expressions do not evaluate to NaN, for each floating-point operation $\boxdot \in \{\boxplus, \boxminus, \boxtimes, \boxdiv\}$ and the corresponding extended real operation $\circ \in \{+, -, \cdot, /\}$, where the entailed inequalities are to be interpreted over $\overline{\mathbb{R}}$:*

$$x \preccurlyeq y \boxdot_{\downarrow} z \implies x \leq y \circ z; \tag{17}$$

moreover, if $x \neq -\infty$,

$$x \preccurlyeq y \boxdot_{\uparrow} z \implies x + \nabla^{\uparrow}(x) < y \circ z; \tag{18}$$

$$x \preccurlyeq y \boxdot_n z \implies \begin{cases} x + \nabla_2^{n-}(x)/2 \leq y \circ z, & \text{if even}(x) \text{ or } x = +\infty; \\ x + \nabla_2^{n-}(x)/2 < y \circ z & \text{if odd}(x); \end{cases} \tag{19}$$

conversely,

$$x \succcurlyeq y \boxdot_{\downarrow} z \implies x + \nabla^{\downarrow}(x) > y \circ z; \tag{20}$$

moreover, if $x \neq +\infty$,

$$x \succcurlyeq y \boxdot_{\uparrow} z \implies x \geq y \circ z; \tag{21}$$

$$x \succcurlyeq y \boxdot_n z \implies \begin{cases} x + \nabla_2^{n+}(x)/2 \geq y \circ z, & \text{if even}(x) \text{ or } x = -\infty; \\ x + \nabla_2^{n+}(x)/2 > y \circ z, & \text{if odd}(x). \end{cases} \tag{22}$$

The proof of Proposition 4 is carried out by applying the inequalities of Proposition 1 to each rounded operation, resulting in a quite long case analysis. It can be found in Appendix B.2.

3.4 Floating-point approximations of constraints on the reals

In this section, we show how possibly complex constraints involving floating-point operations can be approximated directly using floating-point computations, without necessarily using infinite-precision arithmetic.

Without being too formal, let us consider the domain $E_{\mathbb{F}}$ of abstract syntax trees with leafs labelled by constants in \mathbb{F} and internal nodes labeled with a symbol in $\{+, -, \cdot, /\}$ denoting an operation on the reals. While developing propagation algorithms, it is often necessary to deal with inequalities between real numbers and expressions described by such syntax trees. In order to successfully approximate them using the available floating-point arithmetic, we need two functions: $\llbracket \cdot \rrbracket_{\downarrow} : E_{\mathbb{F}} \rightarrow \mathbb{F}$ and $\llbracket \cdot \rrbracket_{\uparrow} : E_{\mathbb{F}} \rightarrow \mathbb{F}$. These functions provide an abstraction of evaluation algorithms that: (a) respect the indicated approximation direction; and (b) are as precise as practical. Point (a) can always be achieved by substituting the real operations with the corresponding floating-point operations rounded in the right direction. For point (b), maximum precision can trivially be achieved whenever the expression involves only one operation; generally speaking, the possibility of efficiently computing a maximally precise (i.e., correctly rounded) result depends on the form of the expression (see, e.g., [27]).

Definition 10 (Evaluation functions) The two partial functions $\llbracket \cdot \rrbracket_\downarrow : E_{\mathbb{F}} \mathbb{F}$ and $\llbracket \cdot \rrbracket_\uparrow : E_{\mathbb{F}} \mathbb{F}$ are such that, for each $e \in \mathbb{F}$ that evaluates on $\overline{\mathbb{R}}$ to a nonzero value,

$$\llbracket e \rrbracket_\downarrow \preceq [e]_\downarrow, \tag{23}$$

$$\llbracket e \rrbracket_\uparrow \succeq [e]_\uparrow. \tag{24}$$

Proposition 5 Let $x \in \mathbb{F}$ be a non-NaN floating point number and $e \in E_{\mathbb{F}}$ an expression that evaluates on $\overline{\mathbb{R}}$ to a nonzero value. The following implications hold:

$$x \geq e \implies x \succeq \llbracket e \rrbracket_\downarrow; \tag{25}$$

$$\text{if } \llbracket e \rrbracket_\downarrow \neq +\infty, x > e \implies x \succeq \text{succ}(\llbracket e \rrbracket_\downarrow); \tag{26}$$

$$x \leq e \implies x \preceq \llbracket e \rrbracket_\uparrow; \tag{27}$$

$$\text{if } \llbracket e \rrbracket_\uparrow \neq -\infty, x < e \implies x \preceq \text{pred}(\llbracket e \rrbracket_\uparrow). \tag{28}$$

In addition, if $\text{pred}(\llbracket e \rrbracket_\uparrow) < e$ (or, equivalently, $\llbracket e \rrbracket_\uparrow = [e]_\uparrow$) we also have

$$x \geq e \implies x \succeq \llbracket e \rrbracket_\uparrow; \tag{29}$$

likewise, if $\text{succ}(\llbracket e \rrbracket_\downarrow) > e$ (or, equivalently, $\llbracket e \rrbracket_\downarrow = [e]_\downarrow$) we have

$$x \leq e \implies x \preceq \llbracket e \rrbracket_\downarrow. \tag{30}$$

The implications of Proposition 5 can be derived from Definition 10 and Proposition 1. Their proof is postponed to Appendix B.2.

4 Propagation for simple arithmetic constraints

In this section we present our propagation procedure for the solution of floating-point constraints obtained from the analysis of programs engaging in IEEE 754 computations.

The general propagation algorithm, which we already introduced in Section 1.2, consists in an iterative procedure that applies the direct and inverse filtering algorithms associated with each constraint, narrowing down the intervals associated with each variable. The process stops when fixed point is reached, i.e., when a further application of any filtering algorithm does not change the domain of any variable.

4.1 Propagation algorithms: definitions

Constraint propagation is a process that prunes the domains of program variables by deleting values that do not satisfy any of the constraints involving those variables. In this section, we will state these ideas more formally.

Let $\boxtimes \in \{\boxplus, \boxminus, \boxtimes, \boxdiv\}$ and $S \subseteq R$. Consider a constraint $x = y \boxtimes_S z$ with $x \in X = [x_\ell, x_u]$, $y \in Y = [y_\ell, y_u]$ with $x \in X = [x_\ell, x_u]$, $y \in Y = [y_\ell, y_u]$ and $z \in Z = [z_\ell, z_u]$.

Direct propagation aims at inferring a narrower interval for variable x , by considering the domains of y and z . It amounts to computing a possibly refined interval for x , $X' = [x'_\ell, x'_u] \subseteq X$, such that

$$\forall r \in S, x \in X, y \in Y, z \in Z : x = y \boxtimes_r z \implies x \in X'. \tag{31}$$

Property (31) is known as the *direct propagation correctness property*.

Of course it is always possible to take $X' = X$, but the objective of constraint propagation is to compute a “small”, possibly the smallest X' enjoying (31), compatibly with the available information. The smallest X' that satisfies (31) is called optimal and is such that

$$\forall X'' \subset X' : \exists r \in S, y \in Y, z \in Z. y \boxplus_r z \notin X'' \tag{32}$$

Property (32) is called the *direct propagation optimality property*.

Inverse propagation, on the other hand, uses the domain of the result x to deduct new domains for the operands, y or z . For the same constraint, $x = y \boxplus_S z$, it means computing a possibly refined interval for y , $Y' = [y'_\ell, y'_u] \subseteq Y$, such that

$$\forall r \in S, x \in X, y \in Y, z \in Z : x = y \boxplus_r z \implies y \in Y' \tag{33}$$

Property (33) is known as the *inverse propagation correctness property*. Again, taking $Y' = Y$ is always possible and sometimes unavoidable. The best we can hope for is to be able to determine the smallest such set, i.e., satisfying

$$\forall Y'' \subset Y : \exists r \in S, y \in Y' \setminus Y'', z \in Z. y \boxplus_r z \notin X \tag{34}$$

Property (34) is called the *inverse propagation optimality property*. Satisfying this last property can be very difficult.

4.2 The Boolean domain for NaN

From now on, we will consider floating-point intervals with boundaries in \mathbb{F} . They allow for capturing the extended numbers in \mathbb{F} only: NaNs (quiet NaNs and signaling NaNs) should be tracked separately. To this purpose, a Boolean domain $\mathcal{N} \stackrel{\text{def}}{=} \{\top, \perp\}$, where \top stands for “may be NaN” and \perp means “cannot be NaN”, can be used and coupled with the arithmetic filtering algorithms.

Let be $x = y \boxplus z$ an arithmetic constraint over floating-point numbers, and (X, NaN_x) , (Y, NaN_y) and (Z, NaN_z) be the variable domains of x , y and z respectively. In practice, the propagation process for such a constraint reaches a fixed point when the combination of refining domains (X', NaN'_x) , (Y', NaN'_y) and (Z', NaN'_z) remains the same obtained in the previous iteration. For each iteration of the algorithm we analyze the NaN domain of all the constraint variables in order to define the next propagator action.

The IEEE 754 Standard [24, Section 7.2] lists all combinations of operand values that yield a NaN result. For the arithmetic operations considered in this paper, NaN is returned if any of the operands is NaN. Moreover, addition and subtraction return NaN when infinities are subtracted (e.g., $+\infty \boxplus -\infty$ or $+\infty \boxminus +\infty$), and also $\pm\infty \boxtimes 0 = \text{NaN}$, $0 \boxtimes \pm\infty = \text{NaN}$, $0 \boxdiv 0 = \text{NaN}$, and $\pm\infty \boxdiv \pm\infty = \text{NaN}$

Thus, direct projections are such that if $\text{NaN}_y = \top$ or $\text{NaN}_z = \top$, then also $\text{NaN}'_x = \top$; indirect projections yield $\text{NaN}'_y = \text{NaN}'_z = \top$ if $\text{NaN}_x = \top$. Moreover, e.g., if $\boxplus = \boxplus$, then the direct projection yields $\text{NaN}'_x = \top$ also if $\pm\infty \in Y$ and $\mp\infty \in Z$, and the indirect one allows for $\pm\infty$ in Y' and $\mp\infty$ in Z' only if $\text{NaN}_x = \top$, and so on for the other operators.

4.3 Filtering algorithms for simple arithmetic constraints

Filtering algorithms for arithmetic constraints are the main focus of this paper. In the next sections, we will propose algorithms realizing optimal *direct* projections and correct *inverse* projections for the addition (\boxplus) and division (\boxdiv) operations. The reader interested in implementing constraint propagation for all four operations can find the algorithms and results for

the missing operations in Appendix A. We report the correctness proofs of the projections for addition in the main text, leaving those for the remaining operations to Appendix B.3.

The filtering algorithms we are about to present are capable of dealing with any set of rounding modes and are designed to distinguish between all different (special) cases in order to be as precise as possible, especially when the variable domains contain symbolic values. Much simpler projections can be designed whenever precision is not of particular concern. Indeed, the algorithms presented in this paper can be considered as the basis for finding a good trade-off between efficiency and the required precision.

4.3.1 Addition

Here we deal with constraints of the form $x = y \boxplus_S z$ with $S \subseteq R$. Let $X = [x_\ell, x_u]$, $Y = [y_\ell, y_u]$ and $Z = [z_\ell, z_u]$.

Thanks to Proposition 2, any set of rounding modes $S \subseteq R$ can be mapped to a pair of “worst-case rounding modes” which, in addition, are never round-to-zero. Therefore, the projection algorithms use the selectors presented in Definition 7 to choose the appropriate worst-case rounding mode, and then operate as if it was the only one in effect, yielding results implicitly valid for the entire set S .

Direct propagation For direct propagation, i.e., the process that infers a new interval for x starting from the interval for y and z , we propose Algorithm 1 and functions da_ℓ and da_u , as defined in Fig. 2. Functions da_ℓ and da_u yield new bounds for interval X . In particular, function da_ℓ gives the new lower bound, while function da_u provides the new upper bound of the interval. Functions da_ℓ and da_u handle all rounding modes and, in order to be as precise as possible, they distinguish between several cases, depending on the values of the bounds of intervals Y and Z . These cases are infinities ($-\infty$ and $+\infty$), zeroes (-0 and $+0$), negative values (\mathbb{R}_-) and positive values (\mathbb{R}_+).

Algorithm 1 Direct projection for addition constraints.

Require: $x = y \boxplus_S z$, $x \in X = [x_\ell, x_u]$, $y \in Y = [y_\ell, y_u]$ and $z \in Z = [z_\ell, z_u]$.

Ensure: $X' \subseteq X$ and $\forall r \in S, x \in X, y \in Y, z \in Z : x = y \boxplus_r z \implies x \in X'$ and $\forall X'' \subset X', \exists r \in S, y \in Y, z \in Z : y \boxplus_r z \notin X''$.

1: $r_\ell := r_\ell(S, y_\ell, \boxplus, z_\ell); r_u := r_u(S, y_u, \boxplus, z_u)$;

2: $x'_\ell := da_\ell(y_\ell, z_\ell, r_\ell); x'_u := da_u(y_u, z_u, r_u)$;

3: $X' := X \cap [x'_\ell, x'_u]$;

It can be proved that Algorithm 1 computes a *correct* and *optimal direct projection*, as stated by its postconditions.

Theorem 1 Algorithm 1 satisfies its contract.

Proof Given the constraint $x = y \boxplus_S z$ with $x \in X = [x_\ell, x_u]$, $y \in Y = [y_\ell, y_u]$ and $z \in Z = [z_\ell, z_u]$, Algorithm 1 sets $X' = [x'_\ell, x'_u] \cap X$; hence, we have $X' \subseteq X$. Moreover, by Proposition 2, for each $y \in Y, z \in Z$ and $r \in S$, we have $y \boxplus_{r_\ell} z \preceq y \boxplus_r z \preceq y \boxplus_{r_u} z$, and because $a \preceq b$ implies $a \leq b$ for any $a, b \in \mathbb{F}$ according to Definition 3, we know that $y \boxplus_{r_\ell} z \leq y \boxplus_r z \leq y \boxplus_{r_u} z$. Thus, by monotonicity of \boxplus , we have $y_\ell \boxplus_{r_\ell} z_\ell \leq y \boxplus_{r_\ell} z \leq y \boxplus_r z \leq y \boxplus_{r_u} z \leq y_u \boxplus_{r_u} z_u$. Therefore, we can focus on finding a lower bound for $y_\ell \boxplus_{r_\ell} z_\ell$ and an upper bound for $y_u \boxplus_{r_u} z_u$.

$da_\ell(y_\ell, z_\ell, r_\ell)$	$-\infty$	\mathbb{R}_-	-0	$+0$	\mathbb{R}_+	$+\infty$
$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$+\infty$
\mathbb{R}_-	$-\infty$	$y_\ell \boxplus_{r_\ell} z_\ell$	y_ℓ	y_ℓ	$y_\ell \boxplus_{r_\ell} z_\ell$	$+\infty$
-0	$-\infty$	z_ℓ	-0	a_1	z_ℓ	$+\infty$
$+0$	$-\infty$	z_ℓ	a_1	$+0$	z_ℓ	$+\infty$
\mathbb{R}_+	$-\infty$	$y_\ell \boxplus_{r_\ell} z_\ell$	y_ℓ	y_ℓ	$y_\ell \boxplus_{r_\ell} z_\ell$	$+\infty$
$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$

$$a_1 = \begin{cases} -0, & \text{if } r_\ell = \downarrow, \\ +0, & \text{otherwise;} \end{cases}$$

$da_u(y_u, z_u, r_u)$	$-\infty$	\mathbb{R}_-	-0	$+0$	\mathbb{R}_+	$+\infty$
$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
\mathbb{R}_-	$-\infty$	$y_u \boxplus_{r_u} z_u$	y_u	y_u	$y_u \boxplus_{r_u} z_u$	$+\infty$
-0	$-\infty$	z_u	-0	a_2	z_u	$+\infty$
$+0$	$-\infty$	z_u	a_2	$+0$	z_u	$+\infty$
\mathbb{R}_+	$-\infty$	$y_u \boxplus_{r_u} z_u$	y_u	y_u	$y_u \boxplus_{r_u} z_u$	$+\infty$
$+\infty$	$-\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$

$$a_2 = \begin{cases} -0, & \text{if } r_u = \downarrow, \\ +0, & \text{otherwise.} \end{cases}$$

Fig. 2 Direct projection of addition: the function da_ℓ (resp., da_u); values for y_ℓ (resp., y_u) on rows, values for z_ℓ (resp., z_u) on columns

Such bounds are given by the functions da_ℓ and da_u of Fig. 2. Almost all of the cases reported in the tables can be trivially derived from the definition of the addition operation in the IEEE 754 Standard [24]; just two cases need further explanation. Concerning the entry of da_ℓ in which $y_\ell = -\infty$ and $z_\ell = +\infty$, note that $z_\ell = +\infty$ implies $z_u = +\infty$. Then for any $y > y_\ell = -\infty$, $y \boxplus +\infty = +\infty$. On the other hand, by the IEEE 754 Standard [24], $-\infty \boxplus +\infty$ is an invalid operation. For the symmetric case, i.e., the entry of da_u in which $y_u = -\infty$ and $z_u = +\infty$, we can reason dually.

We are now left to prove that $\forall X'' \subset X' : \exists r \in S, y \in Y, z \in Z : y \boxplus_r z \notin X''$. Let us focus on the lower bound x'_ℓ , proving that there always exists a $r \in S$ such that $y_\ell \boxplus_r z_\ell = x'_\ell$.

First, consider the cases in which $y_\ell \notin (\mathbb{R}_- \cup \mathbb{R}_+)$ or $z_\ell \notin (\mathbb{R}_- \cup \mathbb{R}_+)$. In these cases, a case analysis proves that $da_\ell(y_\ell, z_\ell, r_\ell)$ is equal to $y_\ell \boxplus_{r_\ell} z_\ell$. Indeed, if either of the operands (say y_ℓ) is $-\infty$ and the other one (say z_ℓ) is *not* $+\infty$, then according to the IEEE 754 Standard we have $y_\ell \boxplus_r z_\ell = -\infty$ for any $r \in R$. Symmetrically, $y_\ell \boxplus_r z_\ell = +\infty$ if one operand is $+\infty$ and the other one is not $-\infty$. If, w.l.o.g., $y_\ell = +\infty$ and $z_\ell = -\infty$, the set X' is non-empty only if $z_u \neq -\infty$, and $y_\ell \boxplus_r z_u = +\infty$ for any $r \in R$.

For the cases in which $y_\ell \in (\mathbb{R}_- \cup \mathbb{R}_+)$ and $z_\ell \in (\mathbb{R}_- \cup \mathbb{R}_+)$ we have $x'_\ell = y_\ell \boxplus_{r_\ell} z_\ell$, by definition of da_ℓ of Fig. 2. Remember that, by Proposition 2, there exists $r \in S$ such that $y_\ell \boxplus_{r_\ell} z_\ell = y_\ell \boxplus_r z_\ell$. Since $y_\ell \in Y$ and $z_\ell \in Z$, we can conclude that for any $X'' \subseteq X'$, $x'_\ell \notin X''$ implies $y_\ell \boxplus_r z_\ell \notin X''$.

An analogous argument allows us to conclude that there exists an $r \in S$ for which the following holds: for any $X'' \subseteq X'$, $x'_u \notin X''$ implies $y_u \boxplus_r z_u \notin X''$. \square

The following example will better illustrate how the tables in Fig. 2 should be used to compute functions da_ℓ and da_u . All examples in this section refer to the IEEE 754 binary single precision format.

Example 1 Assume $Y = [+0, 5]$, $Z = [-0, 8]$, and that the selected rounding mode is $r_\ell = r_u = \downarrow$. In order to compute the lower bound x'_ℓ of X' , the new interval for x , function $\text{da}_\ell(+0, -0, \downarrow)$ is called. These arguments fall in case a_1 , which yields -0 with rounding mode \downarrow . Indeed, when the rounding mode is \downarrow , the sum of -0 and $+0$ is -0 , which is clearly the lowest result that can be obtained with the current choice of Y and Z . For the upper bound x'_u , the algorithm calls $\text{da}_u(5, 8, \downarrow)$. This falls in the case in which both operands are positive numbers ($y_u, z_u \in \mathbb{R}_+$), and therefore $x_u = y_u \boxplus_{r_u} z_u = 13$. In conclusion, the new interval for x is $X' = [-0, 13]$.

If any other rounding mode was selected (say, $r_\ell = r_u = n$), the new interval computed by the projection would have been $X'' = [+0, 13]$.

Inverse propagation For inverse propagation, i.e., the process that infers a new interval for y (or for z) starting from the interval from x and z (x and y , resp.) we define Algorithm 2 and functions ia_ℓ in Fig. 3 and ia_u in Fig. 4, where \equiv indicates the syntactic substitution of expressions. Since the inverse operation of addition is subtraction, note that the values of x and z that minimize y are x_ℓ and z_u ; analogously, the values of x and z that maximize y are x_u and z_ℓ .

When the round-to-nearest rounding mode is in effect, addition presents some nice properties. Indeed, several expressions for lower and upper bounds can be easily computed without approximations, using floating-point operations. In more detail, it can be shown (see the proof of Theorem 2) that when x is subnormal $\nabla_2^{n+}(x)$ and $\nabla_2^{n-}(x)$ are negligible. This allows us to define tight bounds in this case. On the contrary, when the terms $\nabla_2^{n-}(x_\ell)$ and $\nabla_2^{n+}(x_u)$ are non negligible, we need to approximate the values of expressions e_ℓ and e_u . This can always be done with reasonable efficiency [27], but we leave this as an implementation choice, thus accounting for the case when the computation is exact ($\llbracket e_\ell \rrbracket_\downarrow = [e_\ell]$ and $\llbracket e_u \rrbracket = [e_u]$) as well as when it is not ($\llbracket e_\ell \rrbracket > [e_\ell]$ and $\llbracket e_u \rrbracket < [e_u]$).

Algorithm 2 Inverse projection for addition constraints.

Require: $x = y \boxplus_S z$, $x \in X = [x_\ell, x_u]$, $y \in Y = [y_\ell, y_u]$ and $z \in Z = [z_\ell, z_u]$.

Ensure: $Y' \subseteq Y$ and $\forall r \in S$, $x \in X$, $y \in Y$, $z \in Z : x = y \boxplus_r z \implies y \in Y'$.

- 1: $\bar{r}_\ell := \bar{r}_\ell^\ell(S, x_\ell, \boxplus, z_u)$; $\bar{r}_u := \bar{r}_u^\ell(S, x_u, \boxplus, z_\ell)$;
 - 2: $y'_\ell := \text{ia}_\ell(x_\ell, z_u, \bar{r}_\ell)$; $y'_u := \text{ia}_u(x_u, z_\ell, \bar{r}_u)$;
 - 3: **if** $y'_\ell \in \mathbb{F}$ and $y'_u \in \mathbb{F}$ **then**
 - 4: $Y' := Y \cap [y'_\ell, y'_u]$;
 - 5: **else**
 - 6: $Y' := \emptyset$;
 - 7: **end if**
-

The next result assures us that our algorithm computes a *correct inverse projection*, as claimed by its postcondition.

Theorem 2 *Algorithm 2 satisfies its contract.*

Proof Given the constraint $x = y \boxplus_S z$ with $x \in X = [x_\ell, x_u]$, $y \in Y = [y_\ell, y_u]$ and $z \in Z = [z_\ell, z_u]$, Algorithm 2 computes a new and refined domain Y' for variable y .

First, observe that the newly computed interval $[y'_\ell, y'_u]$ is either intersected with the old domain Y , so that $Y' = [y'_\ell, y'_u] \cap Y$, or set to $Y' = \emptyset$. Hence, $Y' \subseteq Y$ holds.

Proposition 2 and the monotonicity of \boxplus allow us to find a lower bound for y by exploiting the constraint $y \boxplus_{\bar{r}_\ell} z_u = x_\ell$, and an upper bound for y by exploiting the constraint

$ia_\ell(x_\ell, z_u, \bar{r}_\ell)$	$-\infty$	\mathbb{R}_-	-0	$+0$	\mathbb{R}_+	$+\infty$
$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
\mathbb{R}_-	unsat.	a_3	x_ℓ	x_ℓ	a_3	$-f_{\max}$
-0	unsat.	$-z_u$	-0	-0	$-z_u$	$-f_{\max}$
$+0$	unsat.	a_4	a_4	a_5	a_4	$-f_{\max}$
\mathbb{R}_+	unsat.	a_3	x_ℓ	x_ℓ	a_3	$-f_{\max}$
$+\infty$	unsat.	$+\infty$	$+\infty$	$+\infty$	a_6	$-f_{\max}$

$$e_\ell \equiv x_\ell + \nabla_2^{n-}(x_\ell)/2 - z_u;$$

$$a_3 = \begin{cases} -0, & \text{if } \bar{r}_\ell = n, \nabla_2^{n-}(x_\ell) = -f_{\min} \text{ and } x_\ell = z_u; \\ x_\ell \boxminus_\uparrow z_u, & \text{if } \bar{r}_\ell = n, \nabla_2^{n-}(x_\ell) = -f_{\min} \text{ and } x_\ell \neq z_u; \\ \llbracket e_\ell \rrbracket_\uparrow, & \text{if } \bar{r}_\ell = n, \text{even}(x_\ell), \nabla_2^{n-}(x_\ell) \neq -f_{\min} \text{ and } \llbracket e_\ell \rrbracket_\uparrow = \llbracket e_\ell \rrbracket_\uparrow; \\ \llbracket e_\ell \rrbracket_\downarrow, & \text{if } \bar{r}_\ell = n, \text{even}(x_\ell), \nabla_2^{n-}(x_\ell) \neq -f_{\min} \text{ and } \llbracket e_\ell \rrbracket_\uparrow > \llbracket e_\ell \rrbracket_\uparrow; \\ \text{succ}(\llbracket e_\ell \rrbracket_\downarrow), & \text{if } \bar{r}_\ell = n, \text{otherwise}; \\ -0, & \text{if } \bar{r}_\ell = \downarrow \text{ and } x_\ell = z_u; \\ x_\ell \boxminus_\uparrow z_u, & \text{if } \bar{r}_\ell = \downarrow \text{ and } x_\ell \neq z_u; \\ \text{succ}(\text{pred}(x_\ell) \boxminus_\downarrow z_u), & \text{if } \bar{r}_\ell = \uparrow; \end{cases}$$

$$(a_4, a_5) = \begin{cases} (\text{succ}(-z_u), +0), & \text{if } \bar{r}_\ell = \downarrow; \\ (-z_u, -0), & \text{otherwise;} \end{cases} \quad a_6 = \begin{cases} +\infty, & \text{if } \bar{r}_\ell = \downarrow; \\ \text{succ}(f_{\max} \boxminus_\downarrow z_u), & \text{if } \bar{r}_\ell = \uparrow; \\ f_{\max} \boxplus_\uparrow (\nabla_2^{n+}(f_{\max})/2 \boxplus_\uparrow z_u), & \text{if } \bar{r}_\ell = n. \end{cases}$$

Fig. 3 Inverse projection of addition: function ia_ℓ

$ia_u(x_u, z_\ell, \bar{r}_u)$	$-\infty$	\mathbb{R}_-	-0	$+0$	\mathbb{R}_+	$+\infty$
$-\infty$	f_{\max}	a_7	$-\infty$	$-\infty$	$-\infty$	unsat.
\mathbb{R}_-	f_{\max}	a_8	x_u	x_u	a_8	unsat.
-0	f_{\max}	a_9	a_{10}	a_9	a_9	unsat.
$+0$	f_{\max}	$-z_\ell$	$+0$	$+0$	$-z_\ell$	unsat.
\mathbb{R}_+	f_{\max}	a_8	x_u	x_u	a_8	unsat.
$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$

$$e_u \equiv x_u + \nabla_2^{n+}(x_u)/2 - z_\ell;$$

$$a_7 = \begin{cases} -\infty, & \text{if } \bar{r}_u = \uparrow; \\ \text{pred}(-f_{\max} \boxplus_\uparrow z_\ell), & \text{if } \bar{r}_u = \downarrow; \\ -f_{\max} \boxplus_\downarrow (\nabla_2^{n-}(-f_{\max}) \boxplus_\downarrow z_\ell); & \text{if } \bar{r}_u = n; \end{cases} \quad (a_9, a_{10}) = \begin{cases} (-z_\ell, +0), & \text{if } \bar{r}_u = \downarrow; \\ (\text{pred}(-z_\ell), -0), & \text{otherwise;} \end{cases}$$

$$a_8 = \begin{cases} +0, & \text{if } \bar{r}_u = n, \nabla_2^{n+}(x_u) = f_{\min} \text{ and } x_u = z_\ell; \\ x_u \boxminus_\downarrow z_\ell, & \text{if } \bar{r}_u = n, \nabla_2^{n+}(x_u) = f_{\min} \text{ and } x_u \neq z_\ell; \\ \llbracket e_u \rrbracket_\downarrow, & \text{if } \bar{r}_u = n, \text{even}(x_u), \nabla_2^{n+}(x_u) \neq f_{\min} \text{ and } \llbracket e_u \rrbracket_\downarrow = \llbracket e_u \rrbracket_\downarrow; \\ \llbracket e_u \rrbracket_\uparrow, & \text{if } \bar{r}_u = n, \text{even}(x_u), \nabla_2^{n+}(x_u) \neq f_{\min} \text{ and } \llbracket e_u \rrbracket_\uparrow < \llbracket e_u \rrbracket_\uparrow; \\ \text{pred}(\llbracket e_u \rrbracket_\uparrow), & \bar{r}_u = n, \text{otherwise}; \\ \text{pred}(\text{succ}(x_u) \boxplus_\uparrow z_\ell), & \text{if } \bar{r}_u = \downarrow; \\ +0, & \text{if } \bar{r}_u = \uparrow \text{ and } x_u = z_\ell; \\ x_u \boxminus_\downarrow z_\ell, & \text{if } \bar{r}_u = \uparrow \text{ and } x_u \neq z_\ell. \end{cases}$$

Fig. 4 Inverse projection of addition: function ia_u

$y \boxplus_{\bar{r}_u} z_\ell = x_u$. We will now prove that the case analyses of functions ia_ℓ , described in Fig. 3, and ia_u , described in Fig. 4, express such bounds correctly.

Concerning the operand combinations in which ia_ℓ takes the value described by the case analysis a_4 , remember that, by the IEEE 754 Standard [24], whenever the sum of two operands with opposite sign is zero, the result of that sum is $+0$ in all rounding-direction attributes except `roundTowardNegative`: in that case the result is -0 . Then, since $z_u \boxplus_{\downarrow} (-z_u) = -0$, when $\bar{r}_\ell = \downarrow$, y_ℓ can safely be set to $\text{succ}(-z_u)$.

As for the case in which ia_ℓ takes one of the values determined by a_5 , the IEEE 754 Standard [24] asserts that $+0 \boxplus_{\downarrow} +0 = +0$, while $-0 \boxplus_{\downarrow} +0 = -0$: the correct lower bound for y is $y'_\ell = +0$, in this case. As we already pointed out, for any other rounding-direction attribute $+0 \boxplus -0 = +0$ holds, which allows us to include -0 in the new domain.

Concerning cases of ia_ℓ that give the result described by the case analysis a_6 , we clearly must have $y = +\infty$ if $\bar{r}_\ell = \downarrow$; if $\bar{r}_\ell = \uparrow$, it should be $y + z_u > f_{\max}$ and thus $y > f_{\max} - z_u$ and, by (28) of Proposition 5, $y \succcurlyeq \text{succ}(f_{\max} \boxminus_{\downarrow} z_u)$. If $\bar{r}_\ell = n$, there are two cases:

$z_u < \nabla_2^{n+}(f_{\max})/2$. In this case, y must be greater than f_{\max} , since $f_{\max} + z_u < f_{\max} + \nabla_2^{n+}(f_{\max})/2$ implies that $f_{\max} \boxplus_n z_u = f_{\max} < +\infty$. Note that in this case $\nabla_2^{n+}(f_{\max})/2 \boxplus_{\uparrow} z_u \geq f_{\min}$, hence $f_{\max} \boxplus_{\uparrow} (\nabla_2^{n+}(f_{\max})/2 \boxplus_{\uparrow} z_u) = +\infty$.

$z_u \geq \nabla_2^{n+}(f_{\max})/2$. Since $\text{odd}(f_{\max})$, for $x_\ell = +\infty$ we need y to be greater than or equal to $f_{\max} + \nabla_2^{n+}(f_{\max})/2 - z_u$. Note that $y \geq f_{\max} + \nabla_2^{n+}(f_{\max})/2 - z_u$ together with

$$[f_{\max} + \nabla_2^{n+}(f_{\max})/2 - z_u]_{\uparrow} = f_{\max} \boxplus_{\uparrow} (\nabla_2^{n+}(f_{\max})/2 \boxplus_{\uparrow} z_u) \tag{35}$$

allows us to apply (29) of Proposition 5, concluding $y \succcurlyeq f_{\max} \boxplus_{\uparrow} (\nabla_2^{n+}(f_{\max})/2 \boxplus_{\uparrow} z_u)$. Equality (35) holds because either the application of ‘ \boxplus_{\uparrow} ’ is exact or the application of ‘ \boxplus_{\uparrow} ’ is exact. In fact, since $z_u = m \cdot 2^e \geq \nabla_2^{n+}(f_{\max})/2 = 2^{e_{\max}-p}$, for some $1 \leq m < 2$, there are two cases: either $e = e_{\max}$ or $e_{\max} - p \leq e < e_{\max}$.

Suppose first that $e = e_{\max}$: we have

$$\begin{aligned} \nabla_2^{n+}(f_{\max})/2 - z_u &= 2^{e_{\max}-p} - m \cdot 2^{e_{\max}} \\ &= -2^{e_{\max}}(m - 2^{-p}), \end{aligned}$$

and thus

$$\nabla_2^{n+}(f_{\max})/2 \boxplus_{\uparrow} z_u = \begin{cases} -2^{e_{\max}}(m - 2^{1-p}), & \text{if } m > 1; \\ -2^{e_{\max}-1}(2 - 2^{1-p}), & \text{if } m = 1. \end{cases}$$

Since if $e = e_{\max}$ the application of ‘ \boxplus_{\uparrow} ’ is not exact, we prove that the application of ‘ \boxplus_{\uparrow} ’ is exact. Hence, if $m > 1$, we prove that

$$\begin{aligned} f_{\max} + (\nabla_2^{n+}(f_{\max})/2 \boxplus_{\uparrow} z_u) &= 2^{e_{\max}}(2 - 2^{1-p}) - 2^{e_{\max}}(m - 2^{1-p}) \\ &= 2^{e_{\max}}(2 - 2^{1-p} - m + 2^{1-p}) \\ &= 2^{e_{\max}}(2 - m) \\ &= 2^{e_{\max}-k} \left(2^k(2 - m) \right) \end{aligned}$$

where $k \stackrel{\text{def}}{=} -\lfloor \log_2(2 - m) \rfloor$. It is worth noting that $2^k(2 - m)$ can be represented by a normalized mantissa; moreover, since $1 \leq k \leq p - 1$, $e_{\min} \leq e_{\max} - k \leq e_{\max}$, hence, $f_{\max} + (\nabla_2^{n+}(f_{\max})/2 \boxplus_{\uparrow} z_u) \in \mathbb{F}$. If, instead, $m = 1$,

$$\begin{aligned} f_{\max} + (\nabla_2^{n+}(f_{\max})/2 \boxplus_{\uparrow} z_u) &= 2^{e_{\max}}(2 - 2^{1-p}) - 2^{e_{\max}-1}(2 - 2^{1-p}) \\ &= (2^{e_{\max}} - 2^{e_{\max}-1})(2 - 2^{1-p}) \\ &= 2^{e_{\max}-1}(2 - 2^{1-p}) \end{aligned}$$

and, also in this case, $f_{\max} + (\nabla_2^{n+}(f_{\max})/2) \boxplus_{\uparrow} z_u \in \mathbb{F}$.

Suppose now that $e_{\max} - p \leq e < e_{\max}$ and let $h \stackrel{\text{def}}{=} e - e_{\max} + p$ so that $0 \leq h \leq p - 1$. In this case we show that the application of ‘ \boxplus_{\uparrow} ’ is exact. Indeed, we have

$$\begin{aligned} \nabla_2^{n+}(f_{\max})/2 - z_u &= 2^{e_{\max} - p} - m \cdot 2^e \\ &= -2^{e_{\max} - p}(m \cdot 2^h - 1) \\ &= -2^{e_{\max} - p + h}(m - 2^{-h}). \end{aligned}$$

If $e = e_{\max} - p$ and $m = 1$, then $h = 0$, $m - 2^{-h} = 0$ and thus $\nabla_2^{n+}(f_{\max})/2 - z_u = 0$. Otherwise, let $k \stackrel{\text{def}}{=} -\lfloor \log_2(m - 2^{-h}) \rfloor$. We have

$$\nabla_2^{n+}(f_{\max})/2 - z_u = -2^{e_{\max} - p + h - k} \left(2^k(m - 2^{-h}) \right),$$

which is an element of \mathbb{F} .

Dual arguments w.r.t. the ones used to justify cases of ia_{ℓ} that give the result described by a_4, a_6 and a_5 can be used to justify the cases of ia_u described by a_9, a_{10} and a_7 .

We now tackle the entries of ia_{ℓ} described by a_3 , and those of ia_u described by a_8 . Exploiting $x \preceq y \boxplus z$ and $x \preceq y \boxminus z$, by Proposition 4, we have

$$y + z \begin{cases} \geq x, & \text{if } \bar{r}_{\ell} = \downarrow; \\ > x + \nabla^{\uparrow}(x) = \text{pred}(x), & \text{if } \bar{r}_{\ell} = \uparrow; \\ \geq x + \nabla_2^{n-}(x)/2, & \text{if } \bar{r}_{\ell} = n \text{ and even}(x); \\ > x + \nabla_2^{n-}(x)/2, & \text{if } \bar{r}_{\ell} = n \text{ and odd}(x). \end{cases}$$

$$y + z \begin{cases} < x + \nabla^{\downarrow}(x) = \text{succ}(x), & \text{if } \bar{r}_u = \downarrow; \\ \leq x, & \text{if } \bar{r}_u = \uparrow; \\ \leq x + \nabla_2^{n+}(x)/2, & \text{if } \bar{r}_u = n \text{ and even}(x); \\ < x + \nabla_2^{n+}(x)/2, & \text{if } \bar{r}_u = n \text{ and odd}(x). \end{cases}$$

The same case analysis gives us

$$y \begin{cases} \geq x - z, & \text{if } \bar{r}_{\ell} = \downarrow; \\ > \text{pred}(x) - z, & \text{if } \bar{r}_{\ell} = \uparrow; \\ \geq x + \nabla_2^{n-}(x)/2 - z, & \text{if } \bar{r}_{\ell} = n \text{ and even}(x); \\ > x + \nabla_2^{n-}(x)/2 - z, & \text{if } \bar{r}_{\ell} = n \text{ and odd}(x); \end{cases}$$

$$y \begin{cases} < \text{succ}(x) - z, & \text{if } \bar{r}_u = \downarrow; \\ \leq x - z, & \text{if } \bar{r}_u = \uparrow; \\ \leq x + \nabla_2^{n+}(x)/2 - z, & \text{if } \bar{r}_u = n \text{ and even}(x); \\ < x + \nabla_2^{n+}(x)/2 - z, & \text{if } \bar{r}_u = n \text{ and odd}(x). \end{cases}$$

We can now exploit the fact that $x \in [x_{\ell}, x_u]$ and $z \in [z_{\ell}, z_u]$ with $x_{\ell}, x_u, z_{\ell}, z_u \in \mathbb{F}$ to obtain, using Proposition 3 and the monotonicity of ‘pred’ and ‘succ’:

$$y \begin{cases} \geq x_{\ell} - z_u, & \text{if } \bar{r}_{\ell} = \downarrow; \\ > \text{pred}(x_{\ell}) - z_u, & \text{if } \bar{r}_{\ell} = \uparrow; \\ \geq x_{\ell} + \nabla_2^{n-}(x_{\ell})/2 - z_u, & \text{if } \bar{r}_{\ell} = n \text{ and even}(x_{\ell}); \\ > x_{\ell} + \nabla_2^{n-}(x_{\ell})/2 - z_u, & \text{if } \bar{r}_{\ell} = n \text{ and odd}(x_{\ell}). \end{cases} \tag{36}$$

$$y \begin{cases} < \text{succ}(x_u) - z_{\ell}, & \text{if } \bar{r}_u = \downarrow; \\ \leq x_u - z_{\ell}, & \text{if } \bar{r}_u = \uparrow; \\ \leq x_u + \nabla_2^{n+}(x_u)/2 - z_{\ell}, & \text{if } \bar{r}_u = n \text{ and even}(x_u); \\ < x_u + \nabla_2^{n+}(x_u)/2 - z_{\ell}, & \text{if } \bar{r}_u = n \text{ and odd}(x_u). \end{cases} \tag{37}$$

We can now exploit Proposition 5 and obtain

$$y'_\ell \stackrel{\text{def}}{=} \begin{cases} -0, & \text{if } \bar{r}_\ell = \downarrow \text{ and } x_\ell = z_u; \\ x_\ell \boxplus_\uparrow z_u, & \text{if } \bar{r}_\ell = \downarrow \text{ and } x_\ell \neq z_u; \\ \text{succ}(\text{pred}(x_\ell) \boxminus_\downarrow z_u), & \text{if } \bar{r}_\ell = \uparrow; \end{cases} \tag{38}$$

$$y'_u \stackrel{\text{def}}{=} \begin{cases} \text{pred}(\text{succ}(x_u) \boxplus_\uparrow z_\ell), & \text{if } \bar{r}_u = \downarrow; \\ +0, & \text{if } \bar{r}_u = \uparrow \text{ and } x_u = z_\ell; \\ x_u \boxminus_\downarrow z_\ell, & \text{if } \bar{r}_u = \uparrow \text{ and } x_u \neq z_\ell. \end{cases} \tag{39}$$

In fact, if $x_\ell = z_u$, then, according to IEEE 754 [24, Section 6.3], for each non-NaN, nonzero and finite $w \in \mathbb{F}$, -0 is the least value for y that satisfies $w = y \boxplus_\downarrow w$. If $x_\ell \neq z_u$, then case (29) of Proposition 5 applies and we have $y \succcurlyeq x_\ell \boxminus_\uparrow z_u$. Suppose now that $\text{pred}(x_\ell) = z_u$, then $\text{pred}(x_\ell) \boxminus_\downarrow z_u \equiv 0$ and $\text{succ}(\text{pred}(x_\ell) \boxminus_\downarrow z_u) = f_{\min}$, coherently with the fact that, for each non-NaN, nonzero and finite $w \in \mathbb{F}$, f_{\min} is the least value for y that satisfies $w = y \boxplus_\uparrow \text{pred}(w)$. If $\text{pred}(x_\ell) \neq z_u$, then case (26) of Proposition 5 applies and we have $y \succcurlyeq \text{succ}(\text{pred}(x_\ell) \boxplus_\downarrow z_u)$. A symmetric argument justifies (39).

For the remaining cases, we first show that when $\nabla_2^{n+}(x) = f_{\min}$,

$$[x_u + \nabla_2^{n+}(x_u)/2 - z_\ell] = [x_u - z_\ell]. \tag{40}$$

The previous equality has the following main consequences: we can perform the computation in \mathbb{F} , that is, we do not need to compute $\nabla_2^{n+}(x)/2$ and, since $[x_u - z_\ell] = \llbracket x_u - z_\ell \rrbracket$, we can apply (30) of Proposition 5, obtaining a tight bound for y'_u .

Let us prove (40). Suppose $\nabla_2^{n+}(x_u) = f_{\min}$, and assume $x_u \neq z_\ell$. There are two cases:

$[x_u - z_\ell]_\downarrow = x_u - z_\ell$: then we have $y \leq [x_u - z_\ell]_\downarrow = x_u - z_\ell$ since the addition of $\nabla_2^{n+}(x_u)/2 = f_{\min}/2$ is insufficient to reach $\text{succ}(x_u - z_\ell)$, whose distance from $x_u - z_\ell$ is at least f_{\min} .

$[x_u - z_\ell]_\downarrow < x_u - z_\ell < [x_u - z_\ell]_\uparrow$: since by Definition 2 every finite floating-point number is an integral multiple of f_{\min} , so are $x_u - z_\ell$ and $[x_u - z_\ell]_\uparrow$. Therefore, again, $y \leq [x_u - z_\ell]_\downarrow$, since the addition of $\nabla_2^{n+}(x_u)/2 = f_{\min}/2$ $x_u - z_\ell$ is insufficient to reach $[x_u - z_\ell]_\uparrow$, whose distance from $x_u - z_\ell$ is at least f_{\min} .

In the case where $x_u = z_\ell$ we have $[x_u + \nabla_2^{n+}(x_u)/2 - z_\ell]_\downarrow = [0 + f_{\min}/2]_\downarrow = +0$, hence (40) holds. As we have already pointed out, this allows us to apply (30) of Proposition 5 to the case $\nabla_2^{n+}(x_u) = f_{\min}$, obtaining the bound $y \preccurlyeq [x_u - z_\ell]_\downarrow$.

Similar arguments can be applied to $\nabla_2^{n-}(x_\ell)$ whenever $\nabla_2^{n-}(x_\ell) = -f_{\min}$ to prove that $[x_\ell + \nabla_2^{n-}(x_\ell)/2 - z_u]_\uparrow = [x_\ell - z_u]_\uparrow$. Then, by (29) of Proposition 5, we obtain $y \succcurlyeq [x_\ell - z_u]_\uparrow$.

When the terms $\nabla_2^{n-}(x_\ell)$ and $\nabla_2^{n+}(x_u)$ are non-negligible, we need to approximate the values of the expressions $e_\ell \stackrel{\text{def}}{=} x_\ell + \nabla_2^{n-}(x_\ell)/2 - z_u$ and $e_u \stackrel{\text{def}}{=} x_u + \nabla_2^{n+}(x_u)/2 - z_\ell$. Hence, we have the cases $\llbracket e_\ell \rrbracket_\uparrow = [e_\ell]_\uparrow$ and $\llbracket e_u \rrbracket_\downarrow = [e_u]_\downarrow$ as well as $\llbracket e_\ell \rrbracket_\uparrow > [e_\ell]_\uparrow$ and $\llbracket e_u \rrbracket_\downarrow < [e_u]_\downarrow$. Thus, when $\llbracket e_u \rrbracket_\downarrow < [e_u]_\downarrow$ by (37) and (27) of Proposition 5 we obtain $y \preccurlyeq \llbracket e_u \rrbracket_\downarrow$, while, when $\llbracket e_\ell \rrbracket_\downarrow > [e_\ell]_\downarrow$ by (37) and (25) of Proposition 5 we obtain $y \succcurlyeq \llbracket e_\ell \rrbracket$. Finally, when $\text{odd}(x_u)$, by (37) and (28) of Proposition 5, we obtain $y \preccurlyeq \text{pred}(\llbracket e_u \rrbracket_\uparrow)$. Dually, when $\text{odd}(x_\ell)$ by (36) and (26) of Proposition 5, we obtain $y \succcurlyeq \text{succ}(\llbracket e_\ell \rrbracket)$.

Thus, for the case $\bar{r}_\ell = n$ we have

$$y'_\ell \stackrel{\text{def}}{=} \begin{cases} -0, & \text{if } \nabla_2^{n-}(x_\ell) = f_{\min} \text{ and } x_\ell = z_u; \\ x_\ell \boxminus_\uparrow z_u, & \text{if } \nabla_2^{n-}(x_\ell) = f_{\min} \text{ and } x_\ell \neq z_u; \\ \llbracket e_\ell \rrbracket_\downarrow, & \text{if } \text{even}(x_\ell), \nabla_2^{n-}(x_\ell) \neq f_{\min} \text{ and } \llbracket e_\ell \rrbracket_\uparrow = \lceil e_\ell \rceil_\uparrow; \\ \llbracket e_\ell \rrbracket_\uparrow, & \text{if } \text{even}(x_\ell), \nabla_2^{n-}(x_\ell) \neq f_{\min} \text{ and } \llbracket e_\ell \rrbracket_\uparrow > \lceil e_\ell \rceil_\uparrow; \\ \text{succ}(\llbracket e_\ell \rrbracket_\downarrow), & \text{otherwise.} \end{cases} \tag{41}$$

whereas, for the case where $\bar{r}_u = n$, we have

$$y'_u \stackrel{\text{def}}{=} \begin{cases} +0, & \text{if } \nabla_2^{n+}(x_u) = f_{\min} \text{ and } x_u = z_\ell; \\ x_u \boxminus_\downarrow z_\ell, & \text{if } \nabla_2^{n+}(x_u) = f_{\min} \text{ and } x_u \neq z_\ell; \\ \llbracket e_u \rrbracket, & \text{if } \text{even}(x_u), \nabla_2^{n+}(x_u) \neq f_{\min} \text{ and } \llbracket e_u \rrbracket_\downarrow = \lfloor e_u \rfloor_\downarrow; \\ \llbracket e_u \rrbracket_\uparrow, & \text{if } \text{even}(x_u), \nabla_2^{n+}(x_u) \neq f_{\min} \text{ and } \llbracket e_u \rrbracket_\uparrow < \lfloor e_u \rfloor_\uparrow; \\ \text{pred}(\llbracket e_u \rrbracket_\uparrow), & \text{otherwise.} \end{cases} \tag{42} \quad \square$$

Example 2 Let $X = [+0, +\infty]$ and $Z = [-\infty, +\infty]$. Regardless of the rounding mode, the calls to functions $\text{ia}_\ell(+0, +\infty, \bar{r}_\ell)$ and $\text{ia}_u(+\infty, -\infty, \bar{r}_u)$ yield $Y' = [-f_{\max}, +\infty]$. Note that $-f_{\max}$ is the lowest value that variable y could take, since there is no value for $z \in Z$ that summed with $-\infty$ gives a value in X . Indeed, if we take $z = +f_{\max}$, then we have $-f_{\max} \boxplus_r +f_{\max} = +0 \in X$ for any $r \in R$. On the other hand, $+\infty$ is clearly the highest value y could take, since $+\infty \boxplus_r z = +\infty \in X$ for any value of $z \in Z \setminus \{-\infty\}$. In this case, our projections yield a more refined result than the competing tool FPSE [10], which computes the wider interval $Y' = [-\infty, +\infty]$.

Example 3 Consider also $X = [1.0, 2.0]$ and $Z = [-1.0 \times 2^{30}, 1.0 \times 2^{30}]$ and $S = \{n\}$. With our inverse projection we obtain $Y = [-1.1 \dots 1 \times 2^{29}, 1.0 \times 2^{30}]$ which is correct but not optimal. For example, pick $y = 1.0 \times 2^{30}$: for $z = -1.0 \times 2^{30}$ we have $y \boxplus_S z = 0$ and $y \boxplus_S z^+ = 64$. By monotonicity of \boxplus_S , for no $z \in [-1.0 \times 2^{30}, 1.0 \times 2^{30}]$ we can have $y \boxplus_S z \in [1.0, 2.0]$.

One of the reasons the inverse projection for addition is not optimal is because floating point numbers present some peculiar properties that are not related in any way to those of real numbers. For interval-based consistency approaches, [29] identified a property of the representation of floating-point numbers and proposed to exploit it in filtering algorithms for addition and subtraction constraints. In [4, 5] some of these authors revised and corrected the Michel and Marre filtering algorithm on intervals for addition/subtraction constraints under the round to nearest rounding mode. A generalization of such algorithm to the all rounding modes should be used to enhance the precision of the classical inverse projection of addition. Indeed, classical and maximum ULP filtering [5] for addition are orthogonal: both should be applied in order to obtain optimal results. Therefore, inverse projections for addition, as the one proposed above, have to be intersected with a filter based on the Michel and Marre property in order to obtain more precise results.

Example 4 Assume, again, $X = [1.0, 2.0]$ and $Z = [-1.0 \times 2^{30}, 1.0 \times 2^{30}]$ and $S = \{n\}$. By applying maximum ULP filtering [5, 29], we obtain the much tighter intervals $Y, Z = [-1.1 \dots 1 \times 2^{24}, 1.0 \times 2^{25}]$. These are actually optimal as $-1.1 \dots 1 \times 2^{24} \boxplus_S 1.0 \times 2^{25} = 1.0 \times 2^{25} \boxplus_S -1.1 \dots 1 \times 2^{24} = 2.0$. This example shows that filtering by maximum ULP can be stronger than our interval-consistency based filtering. However, the opposite phenomenon is also possible. Consider again $X = [1.0, 2.0]$ and $Z = [1.0, 5.0]$. Filtering by

maximum ULP projection gives $Z = [-1.1 \cdots 1 \times 2^{24}, 1.0 \times 2^{25}]$; in contrast, our inverse projection exploits the available information on Z to obtain $Y = [-4, 1.0 \cdots 01]$. As we already stated, our filtering and maximum ULP filtering should both be applied in order to obtain precise results.

Exploiting the commutative property of addition, the refinement Z' of Z can be defined analogously.

4.3.2 Division

In this section we deal with constraints of the form $x = y \boxtimes_S z$ with $S \subseteq R$.

Direct propagation For direct propagation, interval Z is partitioned into the sign-homogeneous intervals $Z_- \stackrel{\text{def}}{=} Z \cap [-\infty, -0]$ and $Z_+ \stackrel{\text{def}}{=} Z \cap [+0, +\infty]$. This is needed because the sign of operand z determines the monotonicity with respect to y , and therefore the interval bounds to be used for propagation depend on it. Hence, once Z has been partitioned into sign-homogeneous intervals, we use the interval Y and $W = Z_-$, to obtain the new interval $[x_\ell^-, x_u^-]$, and Y and $W = Z_+$, to obtain $[x_\ell^+, x_u^+]$. The appropriate bounds for interval propagation are chosen by function τ of Fig. 5. Note that the sign of z is, by construction, constant over interval W . The selected values are then taken as arguments by functions dd_ℓ and dd_u of Fig. 6, which return the correct bounds for the aforementioned new intervals for X . The intervals $X \cap [x_\ell^-, x_u^-]$ and $X \cap [x_\ell^+, x_u^+]$ are eventually joined using convex union, denoted by \uplus , to obtain the refining interval X' .

Algorithm 3 Direct projection for division constraints.

Require: $x = y \boxtimes_S z, x \in X = [x_\ell, x_u], y \in Y = [y_\ell, y_u]$ and $z \in Z = [z_\ell, z_u]$.

Ensure: $X' \subseteq X$ and $\forall r \in S, x \in X, y \in Y, z \in Z : x = y \boxtimes_r z \implies x \in X'$ and $\forall X'' \subset X, \exists r \in S, y \in Y, z \in Z : y \boxtimes_r z \notin X''$.

- 1: $Z_- := Z \cap [-\infty, -0]$;
 - 2: **if** $Z_- = [z_\ell^-, z_u^-] \neq \emptyset$ **then**
 - 3: $W := Z_-$;
 - 4: $(y_L, y_U, w_L, w_U) := \tau(y_\ell, y_u, w_\ell, w_u)$
 - 5: $r_\ell := r_\ell(S, y_L, \boxtimes, w_L); r_u := r_u(S, y_U, \boxtimes, w_U)$;
 - 6: $x_\ell^- := dd_\ell(y_L, w_L, r_\ell); x_u^- := dd_u(y_U, w_U, r_u)$;
 - 7: **else**
 - 8: $[x_\ell^-, x_u^-] := \emptyset$;
 - 9: **end if**
 - 10: $X'_- = X \cap [x_\ell^-, x_u^-]$;
 - 11: $Z_+ := Z \cap [+0, +\infty]$;
 - 12: **if** $Z_+ = [z_\ell^+, z_u^+] \neq \emptyset$ **then**
 - 13: $W := Z_+$;
 - 14: $(y_L, y_U, w_L, w_U) := \tau(y_\ell, y_u, w_\ell, w_u)$
 - 15: $r_\ell := r_\ell(S, y_L, \boxtimes, w_L); r_u := r_u(S, y_U, \boxtimes, w_U)$;
 - 16: $x_\ell^+ := dd_\ell(y_L, w_L, r_\ell); x_u^+ := dd_u(y_U, w_U, r_u)$;
 - 17: **else**
 - 18: $[x_\ell^+, x_u^+] := \emptyset$;
 - 19: **end if**
 - 20: $X'_+ = X \cap [x_\ell^+, x_u^+]$;
 - 21: $X' := X'_- \uplus X'_+$;
-

$$\tau(y_\ell, y_u, w_\ell, w_u) \stackrel{\text{def}}{=} \begin{cases} (y_u, y_\ell, w_\ell, w_u), & \text{if } \text{sgn}(w_u) = \text{sgn}(y_u) = -1; \\ (y_u, y_\ell, w_u, w_\ell), & \text{if } -\text{sgn}(w_u) = \text{sgn}(y_\ell) = 1; \\ (y_u, y_\ell, w_u, w_u), & \text{if } -\text{sgn}(w_u) = -\text{sgn}(y_\ell) = \text{sgn}(y_u) = 1; \\ (y_\ell, y_u, w_\ell, w_u), & \text{if } -\text{sgn}(w_\ell) = \text{sgn}(y_u) = -1; \\ (y_\ell, y_u, w_u, w_\ell), & \text{if } \text{sgn}(w_\ell) = \text{sgn}(y_\ell) = 1; \\ (y_\ell, y_u, w_\ell, w_\ell), & \text{if } \text{sgn}(w_\ell) = -\text{sgn}(y_\ell) = \text{sgn}(y_u) = 1. \end{cases}$$

Fig. 5 Direct projection of division: the function τ ; assumes $\text{sgn}(w_\ell) = \text{sgn}(w_u)$

It can be proved that Algorithm 3 computes a *correct* and *optimal direct projection*, as ensured by its postconditions.

Theorem 3 *Algorithm 3 satisfies its contract.*

Example 5 Consider $Y = [-0, 42]$, $Z = [-3, 6]$ and any value of S . First, Z is split into $Z_- = [-3, -0]$ and $Z_+ = [+0, 6]$. For the negative interval, the third case of $\tau(-0, 42, -3, -0)$ applies, yielding $(y_L, y_U, w_L, w_U) = (42, -0, -0, -0)$. Then, the projection functions are invoked, and we have $\text{dd}_\ell(42, -0, r_\ell) = -\infty$ and $\text{dd}_u(-0, -0, r_u) = +0$, i.e., $[x_\ell^-, x_u^-] = [-\infty, +0]$. For the positive part, we have $\tau(-0, 42, +0, 6) = (-0, 42, +0, +0)$ (sixth case). From the projections we obtain $\text{dd}_\ell(-0, +0, r_\ell) = -0$ and $\text{dd}_u(42, +0, r_u) = +\infty$, and $[x_\ell^+, x_u^+] = [-0, +\infty]$. Finally, $X' = [x_\ell^-, x_u^-] \uplus [x_\ell^+, x_u^+] = [-\infty, +\infty]$.

Inverse propagation first projection The inverse projections of division must be handled separately for each operand. The projection on y is the *first* inverse projection. This case requires, as explained for Algorithm 3, to split Z into the sign-homogeneous intervals $Z_- \stackrel{\text{def}}{=} Z \cap [-\infty, -0]$ and $Z_+ \stackrel{\text{def}}{=} Z \cap [+0, +\infty]$. Then, in order to select the extrema that determine the appropriate lower and upper bound for y , function σ of Fig. 7 is applied. Finally, the functions defined in Figs. 8 and 9 are applied to such extrema.

$\text{dd}_\ell(y_L, w_L, r_\ell)$	$-\infty$	\mathbb{R}_-	-0	$+0$	\mathbb{R}_+	$+\infty$
$-\infty$	$+\infty$	$+\infty$	$+\infty$	$-\infty$	$-\infty$	-0
\mathbb{R}_-	$+0$	$y_L \sqcap_{r_\ell} w_L$	$+\infty$	$-\infty$	$y_L \sqcap_{r_\ell} w_L$	-0
-0	$+0$	$+0$	$+\infty$	-0	-0	-0
$+0$	-0	-0	-0	$+\infty$	$+0$	$+0$
\mathbb{R}_+	-0	$y_L \sqcap_{r_\ell} w_L$	$-\infty$	$+\infty$	$y_L \sqcap_{r_\ell} w_L$	$+0$
$+\infty$	-0	$-\infty$	$-\infty$	$+\infty$	$+\infty$	$+\infty$

$\text{dd}_u(y_U, w_U, r_u)$	$-\infty$	\mathbb{R}_-	-0	$+0$	\mathbb{R}_+	$+\infty$
$-\infty$	$+0$	$+\infty$	$+\infty$	$-\infty$	$-\infty$	$-\infty$
\mathbb{R}_-	$+0$	$y_U \sqcap_{r_u} w_U$	$+\infty$	$-\infty$	$y_U \sqcap_{r_u} w_U$	-0
-0	$+0$	$+0$	$+0$	$-\infty$	-0	-0
$+0$	-0	-0	$-\infty$	$+0$	$+0$	$+0$
\mathbb{R}_+	-0	$y_U \sqcap_{r_u} w_U$	$-\infty$	$+\infty$	$y_U \sqcap_{r_u} w_U$	$+0$
$+\infty$	$-\infty$	$-\infty$	$-\infty$	$+\infty$	$+\infty$	$+0$

Fig. 6 Case analyses for direct propagation of division

$$\sigma(z_\ell, z_u, x_\ell, x_u) \stackrel{\text{def}}{=} \begin{cases} (z_\ell, z_u, x_\ell, x_u), & \text{if } \text{sgn}(z_\ell) = \text{sgn}(x_\ell) = 1; \\ (z_u, z_\ell, x_\ell, x_u), & \text{if } \text{sgn}(z_\ell) = -\text{sgn}(x_u) = 1; \\ (z_u, z_u, x_\ell, x_u), & \text{if } \text{sgn}(z_\ell) = -\text{sgn}(x_\ell) = \text{sgn}(x_u) = 1; \\ (z_u, z_\ell, x_u, x_\ell), & \text{if } \text{sgn}(z_u) = \text{sgn}(x_u) = -1; \\ (z_\ell, z_u, x_u, x_\ell), & \text{if } -\text{sgn}(z_u) = \text{sgn}(x_\ell) = 1; \\ (z_\ell, z_\ell, x_u, x_\ell), & \text{if } -\text{sgn}(z_u) = -\text{sgn}(x_\ell) = \text{sgn}(x_u) = 1. \end{cases}$$

Fig. 7 First inverse projection of division: the function σ ; assumes $\text{sgn}(z_\ell) = \text{sgn}(z_u)$

Algorithm 4 First inverse projection for division constraints.

Require: $x = y \boxtimes_S z$, $x \in X = [x_\ell, x_u]$, $y \in Y = [y_\ell, y_u]$ and $z \in Z = [z_\ell, z_u]$.

Ensure: $Y' \subseteq Y$ and $\forall r \in S, x \in X, y \in Y, z \in Z : x = y \boxtimes_r z \implies y \in Y'$.

```

1:  $Z_- := Z \cap [-\infty, -0]$ ;
2: if  $Z_- = [z_\ell^-, z_u^-] \neq \emptyset$  then
3:    $W := Z_-$ ;
4:    $(w_L, w_U, x_L, x_U) := \sigma(w_\ell, w_u, x_\ell, x_u)$ 
5:    $\bar{r}_\ell := \bar{r}_\ell^f(S, x_L, \boxtimes, w_L)$ ;  $\bar{r}_u := \bar{r}_u^f(S, x_U, \boxtimes, w_U)$ ;
6:    $y_\ell^- := \text{id}_\ell^f(x_L, w_L, \bar{r}_\ell)$ ;  $y_u^- := \text{id}_u^f(x_U, w_U, \bar{r}_u)$ ;
7:   if  $y_\ell^- \in \mathbb{F}$  and  $y_u^- \in \mathbb{F}$  then
8:      $Y'_- = Y \cap [y_\ell^-, y_u^-]$ ;
9:   else
10:     $Y'_- = \emptyset$ ;
11:   end if
12: else
13:    $Y'_- = \emptyset$ ;
14: end if
15:  $Z_+ := Z \cap [+0, +\infty]$ ;
16: if  $Z_+ = [z_\ell^+, z_u^+] \neq \emptyset$  then
17:    $W := Z_+$ ;
18:    $(w_L, w_U, x_L, x_U) := \sigma(w_\ell, w_u, x_\ell, x_u)$ 
19:    $\bar{r}_\ell := \bar{r}_\ell^f(S, x_L, \boxtimes, w_L)$ ;  $\bar{r}_u := \bar{r}_u^f(S, x_U, \boxtimes, w_U)$ ;
20:    $y_\ell^+ := \text{id}_\ell^f(x_L, w_L, \bar{r}_\ell)$ ;  $y_u^+ := \text{id}_u^f(x_U, w_U, \bar{r}_u)$ ;
21:   if  $y_\ell^+ \in \mathbb{F}$  and  $y_u^+ \in \mathbb{F}$  then
22:      $Y'_+ = Y \cap [y_\ell^+, y_u^+]$ ;
23:   else
24:      $Y'_+ = \emptyset$ ;
25:   end if
26: else
27:    $Y'_+ = \emptyset$ ;
28: end if
29:  $Y' := Y'_- \sqcup Y'_+$ ;

```

Example 6 Suppose $X = [-42, +0]$, $Z = [-1.0 \times 2^{100}, -0]$ and $S = \{n\}$. In this case, $Z_- = Z$, and $Z_+ = \emptyset$. We obtain $\sigma(-1.0 \times 2^{100}, -0, -42, +0) = (-1.0 \times 2^{100}, -1.0 \times 2^{100}, +0, -42)$ from the sixth case of σ . Then, $\text{id}_\ell^f(+0, -1.0 \times 2^{100}, n) = (f_{\min} \boxtimes_\uparrow (-1.0 \times 2^{100})) / 2 = -1.0 \times 2^{-50}$, because the lowest value of y_ℓ is obtained when a division by -1.0×2^{100} underflows. Moreover, $\text{id}_u^f(-42, -1.0 \times 2^{100}, n) = 1.0101 \times 2^{105}$. Therefore, the projected interval is $Y' = [-1.0 \times 2^{-50}, 1.0101 \times 2^{105}]$.

$\text{id}_\ell^f(x_L, w_L, \bar{r}_\ell)$	$-\infty$	\mathbb{R}_-	-0	$+0$	\mathbb{R}_+	$+\infty$
$-\infty$	unsat.	a_4	f_{\min}	$-\infty$	$-\infty$	$-\infty$
\mathbb{R}_-	unsat.	a_3^-	f_{\min}	f_{\min}	a_3^+	$-f_{\max}$
-0	$+0$	$+0$	$+0$	f_{\min}	a_7	$-f_{\max}$
$+0$	$-f_{\max}$	a_6	f_{\min}	$+0$	$+0$	$+0$
\mathbb{R}_+	$-f_{\max}$	a_3^-	f_{\min}	f_{\min}	a_3^+	unsat.
$+\infty$	$-\infty$	$-\infty$	$-\infty$	f_{\min}	a_5	unsat.

$$\begin{aligned}
 e_\ell^+ &\equiv (x_L + \nabla_2^-(x_L)/2) \cdot w_L; \\
 a_3^+ &= \begin{cases} \llbracket e_\ell^+ \rrbracket_\uparrow, & \text{if } \bar{r}_\ell = n, \text{ even}(x_L) \text{ and } \llbracket e_\ell^+ \rrbracket_\uparrow = [e_\ell^+]_\uparrow; \\ \llbracket e_\ell^+ \rrbracket_\downarrow, & \text{if } \bar{r}_\ell = n, \text{ even}(x_L) \text{ and } \llbracket e_\ell^+ \rrbracket_\uparrow > [e_\ell^+]_\uparrow; \\ \text{succ}(\llbracket e_\ell^+ \rrbracket_\downarrow), & \text{if } \bar{r}_\ell = n, \text{ otherwise}; \\ x_L \sqsupset_\uparrow w_L, & \text{if } \bar{r}_\ell = \downarrow; \\ \text{succ}(\text{pred}(x_L) \sqsupset_\downarrow w_L), & \text{if } \bar{r}_\ell = \uparrow; \end{cases} \\
 e_\ell^- &\equiv (x_L + \nabla_2^+(x_L)/2) \cdot w_L; \\
 a_3^- &= \begin{cases} \llbracket e_\ell^- \rrbracket_\uparrow, & \text{if } \bar{r}_\ell = n, \text{ even}(x_L) \text{ and } \llbracket e_\ell^- \rrbracket_\uparrow = [e_\ell^-]_\uparrow; \\ \llbracket e_\ell^- \rrbracket_\downarrow, & \text{if } \bar{r}_\ell = n, \text{ even}(x_L) \text{ and } \llbracket e_\ell^- \rrbracket_\uparrow > [e_\ell^-]_\uparrow; \\ \text{succ}(\llbracket e_\ell^- \rrbracket_\downarrow), & \text{if } \bar{r}_\ell = n, \text{ otherwise}; \\ x_L \sqsupset_\uparrow w_L, & \text{if } \bar{r}_\ell = \uparrow; \\ \text{succ}(\text{succ}(x_L) \sqsupset_\downarrow w_L), & \text{if } \bar{r}_\ell = \downarrow; \end{cases} \\
 e_\ell^1 &\equiv (-f_{\max} + \nabla_2^-(f_{\max})/2) \cdot w_L; \\
 a_4 &= \begin{cases} +\infty, & \text{if } \bar{r}_\ell = \uparrow; \\ \text{succ}(-f_{\max} \sqsupset_\downarrow w_L), & \text{if } \bar{r}_\ell = \downarrow; \\ \llbracket e_\ell^1 \rrbracket_\uparrow, & \text{if } \bar{r}_\ell = n \text{ and } [e_\ell^1]_\uparrow = \llbracket e_\ell^1 \rrbracket_\uparrow; \\ \llbracket e_\ell^1 \rrbracket_\downarrow, & \text{if } \bar{r}_\ell = n, \text{ otherwise}; \end{cases} \\
 e_\ell^2 &\equiv (f_{\max} + \nabla_2^+(f_{\max})/2) \cdot w_L; \\
 a_5 &= \begin{cases} +\infty, & \text{if } \bar{r}_\ell = \downarrow; \\ \text{succ}(f_{\max} \sqsupset_\downarrow w_L), & \text{if } \bar{r}_\ell = \uparrow; \\ \llbracket e_\ell^2 \rrbracket_\uparrow, & \text{if } \bar{r}_\ell = n \text{ and } [e_\ell^2]_\uparrow = \llbracket e_\ell^2 \rrbracket_\uparrow; \\ \llbracket e_\ell^2 \rrbracket_\downarrow, & \text{if } \bar{r}_\ell = n, \text{ otherwise}; \end{cases} \\
 (a_6, a_7) &= \begin{cases} (-0, \text{succ}(-f_{\min} \sqsupset_\downarrow w_L)), & \text{if } \bar{r}_\ell = \uparrow; \\ (\text{succ}(f_{\min} \sqsupset_\downarrow w_L), -0), & \text{if } \bar{r}_\ell = \downarrow; \\ ((f_{\min} \sqsupset_\uparrow w_L)/2, (-f_{\min} \sqsupset_\uparrow w_L)/2), & \text{if } \bar{r}_\ell = n. \end{cases}
 \end{aligned}$$

Fig. 8 First inverse projection of division: function id_ℓ^f

The following result assures us that Algorithm 4 computes a *correct first inverse projection*, as ensured by its postcondition.

Theorem 4 *Algorithm 4 satisfies its contract.*

Once again, in order to obtain more precise results in some cases, the first inverse projection for division has to be intersected with a filter based on an extension of the Michel and Marre property originally proposed in [29] and extended to multiplication and division in [5]. Indeed, when interval X does not contain zeroes and interval Z contains zeroes and infinities, the proposed filtering by maximum ULP algorithm is able to derive more precise

$\text{id}_u^f(x_U, w_U, \bar{r}_u)$	$-\infty$	\mathbb{R}_-	-0	$+0$	\mathbb{R}_+	$+\infty$
$-\infty$	$+\infty$	$+\infty$	$+\infty$	$-f_{\min}$	a_9	unsat.
\mathbb{R}_-	f_{\max}	a_8^-	$-f_{\min}$	$-f_{\min}$	a_8^+	unsat.
-0	f_{\max}	a_{12}	$-f_{\min}$	-0	-0	-0
$+0$	-0	-0	-0	$-f_{\min}$	a_{11}	f_{\max}
\mathbb{R}_+	unsat.	a_8^-	$-f_{\min}$	$-f_{\min}$	a_8^+	f_{\max}
$+\infty$	unsat.	a_{10}	$-f_{\min}$	$+\infty$	$+\infty$	$+\infty$

$$\begin{aligned}
 e_u^+ &\equiv (x_U + \nabla_2^{n+}(x_U)/2) \cdot w_U; \\
 a_8^+ &= \begin{cases} \llbracket e_u^+ \rrbracket_{\downarrow}, & \text{if } \bar{r}_u = n, \text{ even}(x_U) \text{ and } \llbracket e_u^+ \rrbracket_{\downarrow} = \llbracket e_u^+ \rrbracket_{\downarrow}; \\ \llbracket e_u^+ \rrbracket_{\uparrow}, & \text{if } \bar{r}_u = n, \text{ even}(x_U) \text{ and } \llbracket e_u^+ \rrbracket_{\downarrow} < \llbracket e_u^+ \rrbracket_{\downarrow}; \\ \text{pred}(\llbracket e_u^+ \rrbracket_{\uparrow}), & \text{if } \bar{r}_u = n, \text{ otherwise;} \\ \text{pred}(\text{succ}(x_U) \square_{\uparrow} w_U), & \text{if } \bar{r}_u = \downarrow; \\ x_U \square_{\downarrow} w_U, & \text{if } \bar{r}_u = \uparrow; \end{cases} \\
 e_u^- &\equiv (x_U + \nabla_2^{n-}(x_U)/2) \cdot w_U; \\
 a_8^- &= \begin{cases} \llbracket e_u^- \rrbracket_{\downarrow}, & \text{if } \bar{r}_u = n, \text{ even}(x_U) \text{ and } \llbracket e_u^- \rrbracket_{\downarrow} = \llbracket e_u^- \rrbracket_{\downarrow}; \\ \llbracket e_u^- \rrbracket_{\uparrow}, & \text{if } \bar{r}_u = n, \text{ even}(x_U) \text{ and } \llbracket e_u^- \rrbracket_{\downarrow} < \llbracket e_u^- \rrbracket_{\downarrow}; \\ \text{pred}(\llbracket e_u^- \rrbracket_{\uparrow}), & \text{if } \bar{r}_u = n, \text{ otherwise;} \\ \text{pred}(\text{pred}(x_U) \square_{\uparrow} w_U), & \text{if } \bar{r}_u = \uparrow; \\ x_U \square_{\downarrow} w_U, & \text{if } \bar{r}_u = \downarrow; \end{cases} \\
 e_u^1 &\equiv (-f_{\max} + \nabla_2^{n-}(-f_{\max})/2) \cdot w_U; \\
 a_9 &= \begin{cases} -\infty, & \text{if } \bar{r}_u = \uparrow; \\ \text{pred}(-f_{\max} \square_{\uparrow} w_U), & \text{if } \bar{r}_u = \downarrow; \\ \llbracket e_u^1 \rrbracket_{\downarrow}, & \text{if } \bar{r}_u = n \text{ and } \llbracket e_u^1 \rrbracket_{\downarrow} = \llbracket e_u^1 \rrbracket_{\downarrow}; \\ \llbracket e_u^1 \rrbracket_{\uparrow}, & \text{if } \bar{r}_u = n, \text{ otherwise;} \end{cases} \\
 e_u^2 &\equiv (f_{\max} + \nabla_2^{n+}(f_{\max})/2) \cdot w_U; \\
 a_{10} &= \begin{cases} -\infty, & \text{if } \bar{r}_u = \downarrow; \\ \text{pred}(f_{\max} \square_{\uparrow} w_U), & \text{if } \bar{r}_u = \uparrow; \\ \llbracket e_u^2 \rrbracket_{\downarrow}, & \text{if } \bar{r}_u = n \text{ and } \llbracket e_u^2 \rrbracket_{\downarrow} = \llbracket e_u^2 \rrbracket_{\downarrow}; \\ \llbracket e_u^2 \rrbracket_{\uparrow}, & \text{if } \bar{r}_u = n, \text{ otherwise;} \end{cases} \\
 (a_{11}, a_{12}) &= \begin{cases} (+0, \text{pred}(-f_{\min} \square_{\uparrow} w_U)), & \text{if } \bar{r}_u = \uparrow; \\ (\text{pred}(f_{\min} \square_{\uparrow} w_U), +0), & \text{if } \bar{r}_u = \downarrow; \\ ((f_{\min} \square_{\downarrow} w_U)/2, (-f_{\min} \square_{\downarrow} w_U)/2), & \text{if } \bar{r}_u = n. \end{cases}
 \end{aligned}$$

Fig. 9 First inverse projection of division: function id_u^f

bounds than the ones obtained with the inverse projection we are proposing. Thus, for division (and for multiplication as well), the indirect projection and filtering by maximum ULP are mutually exclusive: one applies when the other cannot derive anything useful [5].

Example 7 Consider the IEEE 754 single-precision constraint $x = y \square_S z$ with initial intervals $X = [-1.0 \times 2^{-110}, -1.0 \times 2^{-121}]$ and $Y = Z = [-\infty, +\infty]$. When $S = \{n\}$, filtering by maximum ULP results in the possible refinement $Y' = [-1.1 \cdots 1 \times 2^{17}, 1.1 \cdots 1 \times 2^{17}]$, while Algorithm 4 would return the less precise $Y' = [-f_{\max}, f_{\max}]$, with any rounding mode.

Inverse propagation second projection The second inverse projection for division computes a new interval for operand z . For this projection, we need to partition interval X into

sign-homogeneous intervals $X_- \stackrel{\text{def}}{=} X \cap [-\infty, -0]$ and $X_+ \stackrel{\text{def}}{=} X \cap [+0, +\infty]$ since, in this case, it is the sign of X that matters for deriving correct bounds for Z . Once X has been partitioned, we use intervals X_- and Y to obtain the interval $[z_\ell^-, z_u^-]$; intervals X_+ and Y to obtain $[z_\ell^+, z_u^+]$. The new bounds for z are computed by functions id_ℓ^s of Fig. 10 and id_u^s of Fig. 11, after the appropriate interval extrema of Y and $V = X_-$ (or $V = X_+$) have been selected by function τ . The intervals $Z \cap [z_\ell^-, z_u^-]$ and $Z \cap [z_\ell^+, z_u^+]$ will be then joined with convex union to obtain Z' .

Algorithm 5 Second inverse projection for division constraints.

Require: $x = y \square_S z, x \in X = [x_\ell, x_u], y \in Y = [y_\ell, y_u]$ and $z \in Z = [z_\ell, z_u]$.

Ensure: $Z' \subseteq Z$ and $\forall r \in S, x \in X, y \in Y, z \in Z : x = y \square_r z \implies z \in Z'$.

```

1:  $X_- := X \cap [-\infty, -0]$ ;
2: if  $X_- \neq \emptyset$  then
3:    $V := X_-$ ;
4:    $(y_L, y_U, v_L, v_U) := \tau(y_\ell, y_u, v_\ell, v_u)$ 
5:    $\bar{r}_\ell := \bar{r}_r^s(S, v_L, \square, y_L); \bar{r}_u := \bar{r}_u^s(S, v_U, \square, y_U)$ ;
6:    $z_\ell^- := \text{id}_\ell^s(y_L, v_L, \bar{r}_\ell); z_u^- := \text{id}_u^s(y_U, v_U, \bar{r}_u)$ ;
7:   if  $z_\ell^- \in \mathbb{F}$  and  $z_u^- \in \mathbb{F}$  then
8:      $Z'_- = Z \cap [z_\ell^-, z_u^-]$ ;
9:   else
10:     $Z'_- = \emptyset$ ;
11:   end if
12: else
13:    $Z'_- = \emptyset$ ;
14: end if
15:  $X_+ := X \cap [+0, +\infty]$ ;
16: if  $X_+ \neq \emptyset$  then
17:    $V := X_+$ ;
18:    $(y_L, y_U, v_L, v_U) := \tau(y_\ell, y_u, v_\ell, v_u)$ 
19:    $\bar{r}_\ell := \bar{r}_r^s(S, v_L, \square, y_L); \bar{r}_u := \bar{r}_u^s(S, v_U, \square, y_U)$ ;
20:    $z_\ell^+ := \text{id}_\ell^s(y_L, v_L, \bar{r}_\ell); z_u^+ := \text{id}_u^s(y_U, v_U, \bar{r}_u)$ ;
21:   if  $z_\ell^+ \in \mathbb{F}$  and  $z_u^+ \in \mathbb{F}$  then
22:      $Z'_+ = Z \cap [z_\ell^+, z_u^+]$ ;
23:   else
24:     $Z'_+ = \emptyset$ ;
25:   end if
26: else
27:    $Z'_+ = \emptyset$ ;
28: end if
29:  $Z' := Z'_- \sqcup Z'_+$ ;

```

Our algorithm computes a *correct second inverse projection*.

Theorem 5 Algorithm 5 satisfies its contract.

Example 8 Consider $X = [6, +\infty], Y = [+0, 42]$ and $S = \{n\}$. In this case, we only have $X_+ = X$, and $X_- = \emptyset$. With this input, $\tau(+0, 42, 6, +\infty) = (+0, 42, +\infty, 6)$ (case 5).

$\text{id}_\ell^s(y_L, v_L, \bar{r}_\ell)$	$-\infty$	\mathbb{R}_-	-0	$+0$	\mathbb{R}_+	$+\infty$
$-\infty$	$+0$	unsat.	unsat.	$-\infty$	$-f_{\max}$	$-f_{\max}$
\mathbb{R}_-	$+0$	a_3^-	a_4	$-\infty$	a_3^-	a_6
-0	$+0$	f_{\min}	f_{\min}	$-\infty$	$+0$	$+0$
$+0$	$+0$	$+0$	$-\infty$	f_{\min}	f_{\min}	$+0$
\mathbb{R}_+	a_7	a_3^+	$-\infty$	a_5	a_3^+	$+0$
$+\infty$	$-f_{\max}$	$-f_{\max}$	$-\infty$	unsat.	unsat.	$+0$

$$\begin{aligned}
 e_\ell^+ &\equiv y_L / (v_L + \nabla_2^{n+}(v_L) / 2); \\
 a_3^+ &= \begin{cases} \llbracket e_\ell^+ \rrbracket_\uparrow, & \text{if } \bar{r}_\ell = n, \text{ even}(v_L) \text{ and } \llbracket e_\ell^+ \rrbracket_\uparrow = \lceil e_\ell^+ \rceil_\uparrow; \\ \llbracket e_\ell^+ \rrbracket_\downarrow, & \text{if } \bar{r}_\ell = n, \text{ even}(v_L) \text{ and } \llbracket e_\ell^+ \rrbracket_\uparrow > \lceil e_\ell^+ \rceil_\uparrow; \\ \text{succ}(\llbracket e_\ell^+ \rrbracket_\downarrow) & \text{if } \bar{r}_\ell = n, \text{ otherwise;} \\ y_L \boxtimes_\uparrow v_L, & \text{if } \bar{r}_\ell = \uparrow; \\ \text{succ}(y_L \boxtimes_\downarrow \text{succ}(v_L)), & \text{if } \bar{r}_\ell = \downarrow; \end{cases} \\
 e_\ell^- &\equiv y_L / (v_L + \nabla_2^{n-}(v_L) / 2); \\
 a_3^- &= \begin{cases} \llbracket e_\ell^- \rrbracket_\uparrow, & \text{if } \bar{r}_\ell = n, \text{ even}(v_L) \text{ and } \llbracket e_\ell^- \rrbracket_\uparrow = \lfloor e_\ell^- \rfloor_\uparrow; \\ \llbracket e_\ell^- \rrbracket_\downarrow, & \text{if } \bar{r}_\ell = n, \text{ even}(v_L) \text{ and } \llbracket e_\ell^- \rrbracket_\uparrow > \lfloor e_\ell^- \rfloor_\uparrow; \\ \text{succ}(\llbracket e_\ell^- \rrbracket_\downarrow) & \text{if } \bar{r}_\ell = n, \text{ otherwise;} \\ y_L \boxtimes_\uparrow v_L, & \text{if } \bar{r}_\ell = \downarrow; \\ \text{succ}(y_L \boxtimes_\downarrow \text{pred}(v_L)), & \text{if } \bar{r}_\ell = \uparrow; \end{cases} \\
 (a_4, a_5) &= \begin{cases} (+\infty, \text{succ}(y_L \boxtimes_\downarrow f_{\min})), & \text{if } \bar{r}_\ell = \downarrow; \\ (\text{succ}(y_L \boxtimes_\downarrow -f_{\min}), +\infty), & \text{if } \bar{r}_\ell = \uparrow; \\ ((y_L \boxtimes_\uparrow -f_{\min}) \cdot 2, (y_L \boxtimes_\uparrow f_{\min}) \cdot 2), & \text{otherwise;} \end{cases} \\
 e_\ell^1 &\equiv y_L / (f_{\max} + \nabla_2^{n+}(f_{\max}) / 2); \\
 a_6 &= \begin{cases} -0, & \text{if } \bar{r}_\ell = \downarrow; \\ \text{succ}(y_L \boxtimes_\downarrow f_{\max}), & \text{if } \bar{r}_\ell = \uparrow; \\ \llbracket e_\ell^1 \rrbracket_\uparrow, & \text{if } \bar{r}_\ell = n \text{ and } \lceil e_\ell^1 \rceil_\uparrow = \llbracket e_\ell^1 \rrbracket_\uparrow; \\ \llbracket e_\ell^1 \rrbracket_\downarrow, & \text{if } \bar{r}_\ell = n, \text{ otherwise;} \end{cases} \\
 e_\ell^2 &\equiv y_L / (-f_{\max} + \nabla_2^{n-}(-f_{\max}) / 2); \\
 a_7 &= \begin{cases} -0, & \text{if } \bar{r}_\ell = \uparrow; \\ \text{succ}(y_L \boxtimes_\downarrow -f_{\max}), & \text{if } \bar{r}_\ell = \downarrow; \\ \llbracket e_\ell^2 \rrbracket_\uparrow, & \text{if } \bar{r}_\ell = n \text{ and } \lfloor e_\ell^2 \rfloor_\uparrow = \llbracket e_\ell^2 \rrbracket_\uparrow; \\ \llbracket e_\ell^2 \rrbracket_\downarrow, & \text{if } \bar{r}_\ell = n, \text{ otherwise.} \end{cases}
 \end{aligned}$$

Fig. 10 Second inverse projection of division: function id_ℓ^s

Therefore, we obtain $\text{id}_\ell^s(+0, +\infty, n) = +0$, because any number in Y except $+0$ yields $+\infty$ when divided by $+0$. If we compute intermediate values exactly, $\text{id}_u^s(42, 6) = 7$ and the refined interval is $Z' = [+0, 7]$. If not, then $z'_u = 1.110 \dots 01 \times 2^2 = \text{succ}(7)$.

In order to obtain more precise results, the result of our second inverse projection can also be intersected with the interval obtained by the maximum ULP filter proposed in [5]. Indeed, when interval X does not contain zeros and interval Y contains zeros and infinities, the proposed filtering by maximum ULP algorithm is able to derive tighter bounds than those obtained with the inverse projection presented in this work.

Example 9 Consider the IEEE 754 single-precision division constraint $x = y \boxtimes_S z$ with initial intervals $x \in [1.0 \dots 010 \times 2^{110}, 1.0 \times 2^{121}]$ and $Y = Z = [-\infty, +\infty]$. When $S = \{\}$,

$\text{id}_u^s(y_U, v_U, \bar{r}_u)$	$-\infty$	\mathbb{R}_-	-0	$+0$	\mathbb{R}_+	$+\infty$
$-\infty$	f_{\max}	f_{\max}	$+\infty$	unsat.	unsat.	-0
\mathbb{R}_-	a_{11}	a_8^-	$+\infty$	a_9	a_8^-	-0
-0	-0	-0	$+\infty$	$-f_{\min}$	$-f_{\min}$	-0
$+0$	-0	$-f_{\min}$	$-f_{\min}$	$+\infty$	-0	-0
\mathbb{R}_+	-0	a_8^+	a_{10}	$+\infty$	a_8^+	a_{12}
$+\infty$	-0	unsat.	unsat.	$+\infty$	f_{\max}	f_{\max}

$$\begin{aligned}
 e_u^+ &\equiv y_U / (v_U + \nabla_2^{n-}(v_U) / 2); \\
 a_8^+ &= \begin{cases} \llbracket e_u^+ \rrbracket_{\downarrow}, & \text{if } \bar{r}_u = n, \text{ even}(v_U) \text{ and } \llbracket e_u^+ \rrbracket_{\downarrow} = \lceil e_u^+ \rceil_{\downarrow}; \\ \llbracket e_u^+ \rrbracket_{\uparrow}, & \text{if } \bar{r}_u = n, \text{ even}(v_U) \text{ and } \llbracket e_u^+ \rrbracket_{\downarrow} < \lceil e_u^+ \rceil_{\downarrow}; \\ \text{pred}(\llbracket e_u^+ \rrbracket_{\uparrow}), & \text{if } \bar{r}_u = n, \text{ otherwise}; \\ y_U \boxtimes_{\downarrow} v_U, & \text{if } \bar{r}_u = \downarrow; \\ \text{pred}(y_U \boxtimes_{\uparrow} \text{pred}(v_U)), & \text{if } \bar{r}_u = \uparrow; \end{cases} \\
 e_u^- &\equiv y_U / (v_U + \nabla_2^{n+}(v_U) / 2); \\
 a_8^- &= \begin{cases} \llbracket e_u^- \rrbracket_{\downarrow}, & \text{if } \bar{r}_u = n, \text{ even}(v_U) \text{ and } \llbracket e_u^- \rrbracket_{\downarrow} = \lceil e_u^- \rceil_{\downarrow}; \\ \llbracket e_u^- \rrbracket_{\uparrow}, & \text{if } \bar{r}_u = n, \text{ even}(v_U) \text{ and } \llbracket e_u^- \rrbracket_{\downarrow} < \lceil e_u^- \rceil_{\downarrow}; \\ \text{pred}(\llbracket e_u^- \rrbracket_{\uparrow}), & \text{if } \bar{r}_u = n, \text{ otherwise}; \\ y_U \boxtimes_{\downarrow} v_U, & \text{if } \bar{r}_u = \uparrow; \\ \text{pred}(y_U \boxtimes_{\uparrow} \text{succ}(v_U)), & \text{if } \bar{r}_u = \downarrow; \end{cases} \\
 (a_9, a_{10}) &= \begin{cases} (-\infty, \text{pred}(y_U \boxtimes_{\uparrow} -f_{\min})), & \text{if } \bar{r}_u = \uparrow; \\ (\text{pred}(y_U \boxtimes_{\uparrow} f_{\min}), -\infty), & \text{if } \bar{r}_u = \downarrow; \\ ((y_U \boxtimes_{\downarrow} f_{\min}) \cdot 2, (y_U \boxtimes_{\downarrow} -f_{\min}) \cdot 2), & \text{otherwise}; \end{cases} \\
 e_u^1 &\equiv y_U / (-f_{\max} + \nabla_2^{n-}(-f_{\max}) / 2); \\
 a_{11} &= \begin{cases} +0, & \text{if } \bar{r}_u = \uparrow; \\ \text{pred}(y_U \boxtimes_{\uparrow} -f_{\max}), & \text{if } \bar{r}_u = \downarrow; \\ \llbracket e_u^1 \rrbracket_{\downarrow}, & \text{if } \bar{r}_u = n \text{ and } \lceil e_u^1 \rceil_{\downarrow} = \llbracket e_u^1 \rrbracket_{\downarrow}; \\ \llbracket e_u^1 \rrbracket_{\uparrow}, & \text{if } \bar{r}_u = n, \text{ otherwise}; \end{cases} \\
 e_u^2 &\equiv y_U / (f_{\max} + \nabla_2^{n+}(f_{\max}) / 2); \\
 a_{12} &= \begin{cases} +0, & \text{if } \bar{r}_u = \downarrow; \\ \text{pred}(y_U \boxtimes_{\uparrow} f_{\max}), & \text{if } \bar{r}_u = \uparrow; \\ \llbracket e_u^2 \rrbracket_{\downarrow}, & \text{if } \bar{r}_u = n \text{ and } \lceil e_u^2 \rceil_{\downarrow} = \llbracket e_u^2 \rrbracket_{\downarrow}; \\ \llbracket e_u^2 \rrbracket_{\uparrow}, & \text{if } \bar{r}_u = n, \text{ otherwise}. \end{cases}
 \end{aligned}$$

Fig. 11 Second inverse projection of division: function id_u^s

filtering by maximum ULP results in the possible refinement $Z' = [-1.0 \times 2^{18}, 1.0 \times 2^{18}]$, while Algorithm 5 would compute $Z' = [-f_{\max}, f_{\max}]$, regardless of the rounding mode.

5 Experimental evaluation

The main aim of this section is to motivate the need of provably correct filtering algorithms, by highlighting the issues caused by the unsoundness of most available implementations of similar methods.

Table 1 Number of exceptions found by ECLAIR and seVR-fpe on the self-developed benchmarks of [38]

Exception type	ECLAIR	seVR-fpe	Difference
total	135	66	69
overflow	55	26	29
underflow	30	13	17
invalid	47	8	39
divbyzero	3	3	0
false positives	0	15	−15

5.1 Software verification

As we reported in Section 1.3, we implemented our work in the commercial tool ECLAIR. While the initial results on a wide range of self-developed tests looked very promising, we wanted to compare them with the competing tools presented in the literature, in order to better assess the strength of our approach with respect to the state of the art. Unfortunately, most of these tools were either unavailable, or not sufficiently equipped to analyze real-world C/C++ programs. We could, however, do a comparison with the results obtained in [38]. It presents a tool called seVR-fpe, for floating-point exception detection based on symbolic execution and value-range analysis. The same task can be carried out by the constraint-based symbolic model checker we included in ECLAIR. The authors of seVR-fpe tested their tool both on a self-developed benchmark suite and on real-world programs. Upon contacting them, they were unfortunately unable to provide us with more detailed data regarding their analysis of real world programs. This prevents us from doing an in-depth comparison of the tools, since we only know the total number of bugs found, but not their exact nature and location. Data with this level of detail was instead available for (most of) their self-developed benchmarks. The results obtained by running ECLAIR on them are reported in Table 1. ECLAIR could find a number of possible bugs significantly higher than seVR-fpe. As expected, due to the provable correctness of the algorithms employed in ECLAIR, no false positives were detected among the inputs it generated. This confirms the solid results obtainable by means of the algorithms presented in this paper.

5.2 SMT solvers

We compare ECLAIR with several SMT solvers that support floating-point arithmetic by executing them on a benchmark suite devised to test their floating-point theory for soundness. Since our aim is to evaluate filtering algorithms for floating-point addition, subtraction, multiplication and division, we only included tests that do not rely on other theories, such as arrays, bit-vectors and uninterpreted functions, as well as those containing quantifiers. Such features are typical of SMT, and are tackled by techniques which are out of the scope of this paper. The suite is made of a total of 151,432 tests, of which:

- 10,380 are randomly generated tests by Florian Schanda. (random directory from the benchmark suite⁶ used in [12]).

⁶https://github.com/florianschanda/smtlib_schanda, last accessed on October 28th, 2021.

Table 2 Results of the evaluation of SMT-solvers

Solver	Version	Solved	Errors	Unsound	Time (h:m:s)
Colibri	2176	148,766	0	67	01:02:42.03
CVC4	1.8	148,833	0	0	00:09:32.11
ECLAIR	—	148,833	0	0	01:59:28.80
MathSAT ACDL	5.6.5	147,958	0	875	00:07:05.89
z3	4.8.10	148,833	0	0	00:08:38.33

- 11,544 are randomly generated tests by Christoph M. Wintersteiger. (QF_FP/wintersteiger directory from the SMT-LIB benchmark repository.⁷)
- 126,909 tests were generated from the IBM Test Suite for IEEE 754R Compliance⁸ created with FPgen [1].

The experiments were carried out on a high-end laptop with an x86_64 CPU (6 cores @2.20GHz) and 16 GB of RAM, running Ubuntu 20.04. Each solver's version is reported in the table. MathSAT has been executed with the option `-theory.fp.mode=2`, which enables the ACDL-based solver for the floating-point theory.

Results are reported in Table 2. The main observation we can make is that ECLAIR is the only tool based on interval reasoning to be completely sound. On the contrary, Colibri and MathSAT are unsound on numerous tests, even though they did not explicitly report any error. This hinders their use for program verification. On the other hand, bit-blasting based tools CVC4 and Z3 do not present such issues, because their bit-vector encodings for floating-point arithmetic are derived from solid and formally verified circuit designs such as [34]. This demonstrates that the provably-correct filters presented in this paper are needed to achieve reliable implementations of interval-based constraint solving methods.

The execution times seem to be mainly determined by implementation details such as the programming language used. In fact, Colibri and ECLAIR, the slowest tools, were written respectively in ECLIPSe Prolog⁹ and SWI Prolog,¹⁰ while other tools were written in C++.

6 Discussion and conclusion

With the increasing use of floating-point computations in mission- and safety-critical settings, the issue of reliably verifying their correctness has risen to a point in which testing or other informal techniques are not acceptable any more. Indeed, this phenomenon has been fostered by the wide adoption of the IEEE 754 floating-point format, which has significantly simplified the use of floating-point numbers, by providing a precise, sound, and reasonably cross-platform specification of floating-point representations, operations and their semantics. The approach we propose in this paper exploits these solid foundations to enable a

⁷<https://smtlib.cs.uiowa.edu/benchmarks.shtml>, last accessed on October 28th, 2021.

⁸https://www.research.ibm.com/haifa/projects/verification/fpgen/test_suite_download.shtml, last accessed on October 28th, 2021.

⁹<https://eclipseclp.org/>, last accessed on October 28th, 2021.

¹⁰<https://www.swi-prolog.org/>, last accessed on October 28th, 2021.

wide range of floating-point program verification techniques. It is based on the solution of constraint satisfaction problems by means of interval-based constraint propagation, which is enabled by the filtering algorithms we presented. These algorithms cover the whole range of possible floating-point values, including symbolic values, with respect to interval-based reasoning. Moreover, they not only support all IEEE 754 available rounding-modes, but they also allow to take care of uncertainty on the rounding-mode in use. Some important implementation aspects are also taken into account, by allowing both the use of machine floating-point arithmetic for all computations (for increased performance), and of extended-precision arithmetic (for better precision with the round-to-nearest rounding mode). In both cases, correctness is guaranteed, so that no valid solutions can erroneously be removed from the constraint system. This is supported by the extensive correctness proofs of all algorithms and tables, which allow us to claim that neither false positives, nor false negatives may be produced. The experimental evaluation of Section 5 shows that soundness is, indeed, a widespread issue in several floating-point verification tools. Our work provides solid foundations to soundly develop such kind of tools.

Several aspects of the constraint-based verification of floating-point programs remain, however, open problems, both from a theoretical and a practical perspective. As we showed throughout the paper, the filtering algorithms we presented are not optimal, i.e., they may not yield the tightest possible intervals containing all solutions to the constraint system. They must be interleaved with the filtering algorithms of [5], and they may require multiple passes before reaching the maximum degree of variable-domain pruning they are capable of. Therefore, the next possible advance in this direction would be conceiving optimal filtering algorithms, that reduce variable domains to intervals as tight as possible with a single application. This has been achieved in [21], but only for addition.

However, filtering algorithms only represent a significant, but to some extent limited, part of the constraint solving process. Indeed, even an optimally pruned interval may contain values that are not solutions to the constraint system, due to the possible non-linearity thereof. If the framework in use supports multi-intervals, this issue is dealt with by means of labeling techniques: when a constraint-solving process reaches quiescence, i.e., the application of filtering algorithms fails to prune variable domains any further, such intervals are split into two or more sub-intervals, and the process continues on each partition separately. In this context, the main issues are *where* to split intervals, and in *how many* parts. These issues are currently addressed with heuristic labeling strategies. Indeed, significant improvements to the constraint-propagation process could be achieved by investigating better labeling strategies. To this end, possible advancements would include the identification of objective criteria for the evaluation of labeling strategies on floating point-numbers, and the conception of labeling strategies tailored to the properties of constraint systems most commonly generated by numeric programs.

In conclusion, we believe the work presented in this paper can be an extensive reference for the readers interested in realizing applications for formal reasoning on floating-point computations, as well as a solid foundation for further improvements in the state of the art.

Supplementary Information The online version contains supplementary material available at <https://doi.org/10.1007/s10601-021-09322-9>.

Acknowledgements The authors express their gratitude to Roberto Amadini, Isacco Cattabiani and Laura Savino for their careful reading of early versions of this paper.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, 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 licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence 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. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

1. Aharoni, M., Asaf, S., Fournier, L., Koyfman, A., & Nagel, R. (2003). FPGen — a test generation framework for datapath floating-point verification. In *Eighth IEEE international high-level design validation and test workshop* (pp. 17–22). San Francisco: IEEE Computer Society. <https://doi.org/10.1109/HLDVT.2003.1252469>.
2. Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J.D. (2006). *Compilers: principles, techniques, and tools*, 2nd edn. Boston: Addison-Wesley Longman Publishing Co., Inc.
3. Arm Limited: Arm® Architecture Reference Manual, Armv8, for A-profile architecture edn. (2021). <https://developer.arm.com/architectures/cpu-architecture/a-profile/docs>. Last accessed on October 27th, 2021.
4. Bagnara, R., Carlier, M., Gori, R., & Gotlieb, A. (2013). Symbolic path-oriented test data generation for floating-point programs. In *Proceedings of the 6th IEEE international conference on software testing, verification and validation*. Luxembourg City: IEEE Press. <https://doi.org/10.1109/ICST.2013.17>.
5. Bagnara, R., Carlier, M., Gori, R., & Gotlieb, A. (2016). Exploiting binary floating-point representations for constraint propagation. *INFORMS Journal on Computing*, 28(1), 31–46. <https://doi.org/10.1287/ijoc.2015.0663>.
6. Bagnara, R., Chiari, M., Gori, R., & Bagnara, A. (2021). A practical approach to verification of floating-point C/C++ programs with math.h/cmath functions. *ACM Transactions on Software Engineering and Methodology*, 30(1), 9:1–9:53. <https://doi.org/10.1145/3410875>.
7. Barr, E. T., Vo, T., Le, V., & Su, Z. (2013). Automatic detection of floating-point exceptions. In *The 40th annual ACM SIGPLAN-SIGACT symposium on principles of programming languages, POPL '13, Rome, Italy - January 23 - 25, 2013* (pp. 549–560). <https://doi.org/10.1145/2429069.2429133>.
8. Barrett, C. W., Conway, C. L., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., & Tinelli, C. (2011). CVC4. In *Computer aided verification, proceedings of the 23rd international conference (CAV 2011), Lecture notes in computer science*, (Vol. 6806 pp. 171–177). Snowbird: Springer. https://doi.org/10.1007/978-3-642-22110-1_14.
9. Barrett, C. W., & Tinelli, C. (2018). Satisfiability modulo theories. In E. M. Clarke, T. A. Henzinger, H. Veith, & R. Bloem (Eds.) *Handbook of model checking* (pp. 305–343). Springer. https://doi.org/10.1007/978-3-319-10575-8_11.
10. Botella, B., Gotlieb, A., & Michel, C. (2006). Symbolic execution of floating-point computations. *Software Testing, Verification and Reliability*, 16(2), 97–121. <https://doi.org/10.1002/strv.333>.
11. Brain, M., D'Silva, V., Griggio, A., Haller, L., & Kroening, D. (2014). Deciding floating-point logic with abstract conflict driven clause learning. *Formal Methods in System Design*, 45(2), 213–245. <https://doi.org/10.1007/s10703-013-0203-7>.
12. Brain, M., Schanda, F., & Sun, Y. (2019). Building better bit-blasting for floating-point problems. In *Tools and algorithms for the construction and analysis of systems, proceedings of the 25th international conference (TACAS 2019), Part I, lecture notes in computer science*, (Vol. 11427 pp. 79–98). Springer. https://doi.org/10.1007/978-3-030-17462-0_5.
13. Brain, M., Tinelli, C., & Rümmer, P. (2015). T-wahl: An automatable formal semantics for IEEE-754 floating-point arithmetic. In *22nd IEEE symposium on computer arithmetic (ARITH 2015)* (pp. 160–167). Lyon: IEEE. <https://doi.org/10.1109/ARITH.2015.26>.
14. Cimatti, A., Griggio, A., Schaafsma, B. J., & Sebastiani, R. (2013). The mathSAT5 SMT solver. In *Tools and algorithms for the construction and analysis of systems, proceedings of the 19th international conference (TACAS 2013), lecture notes in computer science*, (Vol. 7795 pp. 93–107). Springer. https://doi.org/10.1007/978-3-642-36742-7_7.

15. Clarke, L. A., & Richardson, D. J. (1985). Applications of symbolic evaluation. *Journal of Systems and Software*, 5(1), 15–35. [https://doi.org/10.1016/0164-1212\(85\)90004-4](https://doi.org/10.1016/0164-1212(85)90004-4).
16. Cousot, P., & Cousot, R. (1977). Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the fourth annual ACM symposium on principles of programming languages* (pp. 238–252). Los Angeles: ACM Press. <https://doi.org/10.1145/512950.512973>.
17. Cuyt, A., Kuterna, P., Verdonk, B., & Verschaeren, D. (2002). Underflow revisited. *CALCOLO*, 39(3), 169–179. <https://doi.org/10.1007/s100920200003>.
18. de Moura, L. M., & Björner, N. (2008). Z3: an efficient SMT solver. In *Tools and algorithms for the construction and analysis of systems, proceedings of the 14th international conference (TACAS 2008), lecture notes in computer science*, (Vol. 4963 pp. 337–340). Springer. https://doi.org/10.1007/978-3-540-78800-3_24.
19. Delmas, D., Goubault, E., Putot, S., Souyris, J., Tekkal, K., & Védrine, F. (2009). Towards an industrial use of FLUCTUAT on safety-critical avionics software. In *Formal methods for industrial critical systems, proceedings of the 14th international workshop (FMICS 2009), lecture notes in computer science*, (Vol. 5825 pp. 53–69). Eindhoven: Springer. https://doi.org/10.1007/978-3-642-04570-7_6.
20. Fiedler, G. (2021). Floating point determinism. Gaffer on games blog, February 24, 2019. https://gafferongames.com/post/floating_point_determinism/ (2010) Last accessed on October 28th.
21. Gallois-Wong, D., Boldo, S., & Cuoq, P. (2020). Optimal inverse projection of floating-point addition. *Numerical Algorithms*, 83(3), 957–986. <https://doi.org/10.1007/s11075-019-00711-z>.
22. Gotlieb, A., Botella, B., & Rueher, M. (1998). Automatic test data generation using constraint solving techniques. In *Proceedings of the 1998 ACM SIGSOFT international symposium on software testing and analysis, ISSTA '98* (pp. 53–62). New York: ACM. <https://doi.org/10.1145/271771.271790>.
23. Gotlieb, A., Botella, B., & Rueher, M. (2000). A CLP framework for computing structural test data. In *Computational logic — CL 2000: first international conference london, UK, July 24–28, 2000 Proceedings* (pp. 399–413). Springer. https://doi.org/10.1007/3-540-44957-4_27.
24. The Institute of Electrical and Electronics Engineers, Inc.: IEEE Standard for Floating-Point Arithmetic, IEEE Std 754-2008 (revision of IEEE Std 754-1985) edn. (2008). Available at <http://ieeexplore.ieee.org/servelet/opac?punumber=4610933>.
25. Intel Corporation. Intel® 64 and IA-32 Architectures Software Developer Manuals, 325462-075us edn. (2021). <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html>. Last accessed on October 27th, 2021.
26. King, J. C. (1976). Symbolic execution and program testing. *Communications of the ACM*, 19(7), 385–394. <https://doi.org/10.1145/360248.360252>.
27. Kornerup, P., Lefevre, V., Louvet, N., & Muller, J.M. (2009). On the computation of correctly-rounded sums. In *Proceedings of the 19th IEEE symposium on computer arithmetic (ARITH 2009)* (pp. 155–160). Portland. <https://doi.org/10.1109/ARITH.2009.16>.
28. Marre, B., Bobot, F., & Chihani, Z. (2017). Real behavior of floating point numbers. In *Proceedings of the 15th international workshop on satisfiability modulo theories, CEUR workshop proceedings*, (Vol. 1889 pp. 50–62).
29. Marre, B., & Michel, C. (2010). Improving the floating point addition and subtraction constraints. In D. Cohen (Ed.) *Proceedings of the 16th international conference on principles and practice of constraint programming (CP 2010), lecture notes in computer science*, (Vol. 6308 pp. 360–367). St. Andrews: Springer. https://doi.org/10.1007/978-3-642-15396-9_30.
30. Michel, C. (2002). Exact projection functions for floating point number constraints. In *Proceedings of the 7th international symposium on artificial intelligence and mathematics*. Fort Lauderdale.
31. Michel, C., Rueher, M., & Lebbah, Y. (2001). Solving constraints over floating-point numbers. In *Principles and practice of constraint programming - CP 2001, 7th international conference, CP 2001, paphos, cyprus, november 26 - december 1, 2001, proceedings* (pp. 524–538). https://doi.org/10.1007/3-540-45578-7_36.
32. Miné, A. (2004). Relational abstract domains for the detection of floating-point run-time errors. In *Programming languages and systems, proceedings of the 13th european symposium on programming (ESOP 2004), lecture notes in computer science*, (Vol. 2986 pp. 3–17). Barcelona: Springer. https://doi.org/10.1007/978-3-540-24725-8_2.
33. Monniaux, D. (2008). The pitfalls of verifying floating-point computations. *ACM Transactions on Programming Languages and Systems*, 30(3), 12:1–12:41. <https://doi.org/10.1145/1353445.1353446>.
34. Müller, S. M., & Paul, W. J. (2000). Computer architecture - complexity and correctness springer. <https://doi.org/10.1007/978-3-662-04267-0>.
35. Rump, S. M. (2013). Accurate solution of dense linear systems, Part II: Algorithms using directed rounding. *Journal of Computational and Applied Mathematics*, 242, 185–212. <https://doi.org/10.1016/j.cam.2012.09.024>.

36. Rump, S. M., & Ogita, T. (2007). Super-fast validated solution of linear systems. *Journal of Computational and Applied Mathematics*, 199(2), 199–206. Special Issue on Scientific Computing, Computer Arithmetic, and Validated Numerics (SCAN 2004).
37. Watte, J. (2021). Floating point determinism. Gamedev.net Forum, June 30, 2008. <https://www.gamedev.net/forums/topic/499435-floating-point-determinism/> (2008) Last accessed on October 28th.
38. Wu, X., Li, L., & Zhang, J. (2017). Symbolic execution with value-range analysis for floating-point exception detection. In *24th asia-pacific software engineering conference, APSEC 2017, Nanjing, China, December 4–8, 2017* (pp. 1–10). <https://doi.org/10.1109/APSEC.2017.6>.

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.