P1788/D7.1
Draft Standard For Interval Arithmetic

John Pryce and Christian Keil, Technical Editors

Frontmatter

Required IEEE items to be added:

– Draft copyright statements
– Title
– Abstract and keywords
– Committee lists
– Acknowledgments (after the Introduction)

⚠ To the P1788 working group reader.

General. Passages in this color are my editorial comments: mostly asking for answers to or debate on a question; or giving my opinion; or noting changes made.

Introduction

This introduction explains some of the alternative interpretations, and sometimes competing objectives, that influenced the design of this standard, but is not part of the standard.

Mathematical context. Interval computation is a collaboration between human programmer and machine infrastructure which, correctly done, produces mathematically proven numerical results about continuous problems—for instance, rigorous bounds on the global minimum of a function or the solution of a differential equation. It is part of the discipline of "constructive real analysis". In the long term, the results of such computations may become sufficiently trusted to be accepted as contributing to legal decisions. The machine infrastructure acts as a body of theorems on which the correctness of an interval algorithm relies, so it must be made as reliable as is practical. In its logical chain are many links—hardware, underlying floating-point system, etc.—over which this standard has no control. The standard aims to strengthen one specific link, by defining interval objects and operations that are theoretically well-founded and practical to implement.

This document uses the standard notation $[a, b]$ for "the interval between numbers a and b", with various detailed meanings depending on the underlying theory. The "classical" interval arithmetic (IA) of R.A. Moore [**5**] uses only bounded, closed, nonempty intervals in the real numbers $\mathbb{R}$—that is, $[a, b] = \{\, x \in \mathbb{R} \mid a \le x \le b \,\}$ where $a, b \in \mathbb{R}$ with $a \le b$. So, for instance, division by an interval containing 0 is not defined in it. It was agreed early on that this standard should strictly extend classical IA in virtue of allowing an interval to be unbounded or empty.

Beyond this, various extensions of classical IA were considered. One choice that distinguishes between theories is: Are arithmetic operations purely algebraic, or do they involve topology? An example of the latter is containment set (cset) theory [**8**], which extends functions over the reals to functions over the extended reals, e.g. $\sin(+\infty)$ is the set of all possible limits of $\sin x$ as $x \to +\infty$, which is $[-1, 1]$. The complications of this were deemed to outweigh the advantages, and it was agreed that operations should be purely algebraic.

Another choice is: Is an interval a set—a subset of the number line—or is it something different? The most widely used forms of IA are *set-based* and define an interval to be a set of real numbers. They have established software to find validated solutions of linear and nonlinear algebraic equations, optimization problems, differential equations, etc.

However *Kaucher* IA and the nearly equivalent *modal* IA have significant applications. In the former an interval is formally a pair (a, b) of real numbers, which for $a \le b$ is "proper" and identified with the normal interval $\{\, x \in \mathbb{R} \mid a \le x \le b \,\}$, and for $a > b$ is "improper". In the latter, an interval is a pair (X, Q) where X is a normal interval and Q is a quantifier, either $\exists$ or $\forall$. At the time of writing it finds commercial use in the graphics rendering industry. Both forms are referred to as Kaucher IA henceforth.

In view of their significance it was decided to support both set-based and Kaucher IA. Because of their different mathematical bases this led to the concept of *flavors* (see Clause 7). A flavor is a version of IA that extends classical IA in a precisely defined sense, such that when only classical intervals and restricted operations are used (avoiding, e.g., division by an interval containing zero), all flavors produce the same results, at the mathematical level and also—up to roundoff—in finite precision.

Currently the standard incorporates two flavors, set-based and Kaucher. Others could be added, though there are no current plans to do so. E.g., csets could be a flavor, since they also extend classical IA in the defined sense.

To minimize complexity, the standard for each flavor is presented as a separate sub-document within the overall standard, readable as a self-contained unit without reference to other flavors, and with common clauses that specify how a flavor extends classical IA.

The set-based flavor is presented first, on the grounds that it is relatively easy to grasp, easy to teach, and easy to interpret in the context of real-world applications. In this theory:

– Intervals are sets.
– They are subsets of the set $\mathbb{R}$ of real numbers. At the mathematical level (Level 1 in the structure defined in §**??**) they are precisely all topologically closed and connected subsets of $\mathbb{R}$. The finite-precision level (Level 2), uses the notion of an interval type, which is a finite set of Level 1 intervals.
– The interval version of an elementary function such as $\sin x$ is essentially the natural algebraic extension to sets of the corresponding pointwise function on real numbers.

Fuzzy sets, like intervals, are a way to handle uncertain knowledge, and the two topics are related. However, considering this relation was beyond the scope of this project.

Specification Levels. The 754-2008 standard describes itself as layered into four Specification Levels. To manage complexity, the present standard uses a corresponding structure. It deals mainly with Level 1, of mathematical *interval theory*, and Level 2, the finite set of *interval datums* in terms of which finite-precision interval computation is defined. It has some concern with Level 3, of *representations* of intervals as data structures; and none with Level 4, of *bit strings* and memory.

There is another important player: the programming language. It was a recognized omission of IEEE-754-1985 that it specified individual operations but not how they should be used in expressions. Optimizing compilers have, since well before that standard, used clever transformations so that it is impossible to know the precisions used and the roundings performed while evaluating an expression, or whether the compiler has even "optimized away" $(1.0 + x) - 1.0$ to become simply x. IEEE-754-2008 specifies this by placing requirements on how operations should be used in expressions, though as of this writing, few programming languages have adopted that.

The lack of any restrictions is also a problem for intervals. Thus the standard makes requirements and recommendations on language implementations, thereby defining the notion of a standard-conforming implementation of intervals within a language.

The language does not constitute a fifth level in some linear sequence; from the user's viewpoint most current languages sit above datum level 2, alongside theory level 1, as a practical means to implement interval algorithms by manipulating Level 2 entities (though most languages have influence on Levels 3 and 4 also). This standard extends them to provide an instantiation of level 2 entities.

The Fundamental Theorem. Moore's [**5**] Fundamental Theorem of Interval Arithmetic (FTIA) is central to interval computation. Roughly, it says as follows. Let f be an *explicit arithmetic expression*—that is, it is built from finitely many elementary functions (arithmetic operations) such as $+, -, \times, \div, \sin, \exp, \ldots$, with no non-arithmetic operations such as intersection, so that it defines a real function $f(x_1, \ldots, x_n)$. Then evaluating f "in interval mode" over any interval inputs $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$ is guaranteed to give an enclosure of the range of f over those inputs.

A version of the FTIA holds in all variants of interval theory, but with varying hypotheses and conclusions. In the context of this standard, an expression should be evaluated entirely in one flavor, and inferences made strictly from that flavor's FTIA; otherwise, a user may believe an FTIA holds in a case where it does not, with possibly serious effects in applications. As stated, the FTIA is about the mathematical level. Moore's achievements were to see that "outward rounding" makes the FTIA hold also in finite precision, and to follow through the consequences. An advantage of the level structure used by the standard is that the mapping between levels 1 and 2 defines a framework where it is easily proved that

The finite-precision FTIA holds in any conforming implementation.

Generally it can only be determined *a posteriori* whether the conditions for any version of the FTIA hold; this is an important application of the standard's *decoration system.*

For each flavor in the standard, its subdocument must state precisely the form of the FTIA it obeys, both at the mathematical level 1 and at the finite-precision level 2.

Operations.

There are several interpretations of *evaluation outside an operation's domain* and *operations as relations rather than functions*. This includes classical alternative meanings of division by an interval containing zero, or square root of an interval containing negative values. To illustrate the different interpretations, consider $\boldsymbol{y} = \sqrt{\boldsymbol{x}}$ where $\boldsymbol{x} = [-1, 4]$.

(1) In *optimization*, when computing lower bounds on the objective function, it is generally appropriate to return the result $\boldsymbol{y} = [0, 2]$, and ignore the fact that $\sqrt{\cdot}$ has been applied to negative elements of $\boldsymbol{x}$.

(2) In applications where one must check the hypotheses of a *fixed point theorem* are satisfied (such as solving differential equations):
   (a) one may need to be sure that the function is defined and continuous on the input and, hence, report an illegal argument when, as in the above case, this fails; or
   (b) one may need the result $\boldsymbol{y} = [0, 2]$, but must flag the fact that $\sqrt{\cdot}$ has been evaluated at points where it is undefined or not continuous.

(3) In *constraint propagation*, the equation is often to be interpreted as: find an interval enclosing all y such that $y^2 = x$ for some $x \in [-1, 4]$. In this case the answer is $[-2, 2]$.

The standard provides means to meet these diverse needs, while aiming to preserve clarity and efficiency. A language might achieve this by binding one of the above three interpretations—usually some variant of (2)—to its built-in operations, and providing the others as library procedures.

In the context of flavors, a key idea is that of *common operation instances*: those elementary interval calculations that at the mathematical level are required to give the same result in all flavors. For example $[1, 2]/[3, 4] = [1/4, 2/3]$ is common, while division by an interval containing zero is not common.

Decorations.

Many interval algorithms are only valid if certain mathematical conditions are satisfied: for instance one may need to know that a function, defined by an expression, is everywhere continuous on a box in $\mathbb{R}^n$ defined by n input intervals $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$. The IEEE 754 model of global flags to record events such as division by zero was considered inadequate in an era of massively parallel processing. In this standard, such events are recorded locally by *decorations*.

A *decorated interval* is an ordinary interval tagged with a few bits that encode the decoration, and record while evaluating an expression, e.g., "each elementary function was defined and continuous on its inputs"—which implies the same for the function defined by the whole expression. This makes possible a rigorous check of properties such as listed in item (2) of §. A small number of decorations is provided, designed for efficient propagation of such property information.

Care was taken to meet different user needs. *Bare* (undecorated) intervals are available for simple use without validity checks. *Decorated* intervals are recommended for serious programming, but suffer the "17-byte problem": a typical bare interval stored as two doubles takes up 16 bytes, so a decorated one needs at least 17 bytes. With large problems on typical machine architectures this may cause inefficiencies—in data throughput if storing 17-byte data structures, or in storage if one pads the structure to, say, 32 bytes. Hence an optional *compressed* decorated interval scheme is specified, for advanced use. It aims to give the speed of 16-byte objects, at a cost in flexibility but supporting applications such as checking whether a function is defined and continuous on its inputs.

Contents

CHAPTER 1

General Requirements

1. Overview

1.1. Scope. This standard specifies basic interval arithmetic (IA) operations selecting and following one of the commonly used mathematical interval models. This standard supports the IEEE-754-2008 floating point formats of practical use in interval computations. Exception conditions are defined and standard handling of these conditions is specified. Consistency with the model is tempered with practical considerations based on input from representatives of vendors and owners of existing systems.

The standard provides a layer between the hardware and the programming language levels. It does not mandate that any operations be implemented in hardware. It does not define any realization of the basic operations as functions in a programming language.

1.2. Purpose. The aim of the standard is to improve the availability of reliable computing in modern hardware and software environments by defining the basic building blocks needed for performing interval arithmetic. There are presently many systems for interval arithmetic in use; lack of a standard inhibits development, portability; ability to verify correctness of codes.

1.3. Inclusions. This standard specifies

– Types for interval data based on underlying numeric formats, with a special class of type derived from IEEE 754 floating point formats.
– Constructors for intervals from numeric and character sequence data.
– Addition, subtraction, multiplication, division, fused multiply add, square root; other interval-valued operations for intervals.
– Midpoint, radius and other numeric functions of intervals.
– Interval comparison relations.
– Required elementary functions.
– Conversions between different interval types.
– Conversions between interval types and external representations as character sequences.
– Interval-related exceptions and their handling.

1.4. Exclusions. This standard does not specify

– Which numeric formats supported by the underlying system shall have an associated interval type.
– Details of how an implementation represents intervals at the level of programming language data types, or bit patterns.

1.5. Word usage.

In this standard three words are used to differentiate between different levels of requirements and optionality, as follows:

– **may** indicates a course of action permissible within the limits of the standard with no implied preference ("may" means "is permitted to");
– **shall** indicates mandatory requirements strictly to be followed in order to conform to the standard and from which no deviation is permitted ("shall" means "is required to");
– **should** indicates that among several possibilities, one is recommended as particularly suitable, without mentioning or excluding others; or that a certain course of action is preferred but not necessarily required; or that (in the negative form) a certain course of action is deprecated but not prohibited ("should" means "is recommended to").

Further:

– **optional** indicates features that may be omitted, but if provided shall be provided exactly as specified.
– **might** indicates the possibility of a situation that could occur, with no implication of the likelihood of that situation ("might" means "could possibly");
– **see** followed by a number is a cross-reference to the clause or subclause of this standard identified by that number;
– **comprise** indicates members of a set are exactly those objects having some property, e.g. "the set of mathematical intervals comprises the closed, connected subsets of $\mathbb{R}$"; an unqualified **consist of** merely asserts all members of a set have some property, e.g. "a binary floating point format consists of numbers with a terminating binary representation". "Comprises" means "consists exactly of".
– **Note** and **Example** introduce text that is informative (is not a requirement of this standard).

1.6. The meaning of conformance. §3 lists the requirements on a conforming implementation in summary form, with references to where these are stated in detail.

1.7. Programming environment considerations.
This standard does not define all aspects of a conforming programming environment. Such behavior should be defined by a programming language definition supporting this standard, if available; otherwise by a particular implementation. Some programming language specifications might permit some behaviors to be defined by the implementation.

Language-defined behavior should be defined by a programming language standard supporting this standard. Standards for languages intended to reproduce results exactly on all platforms are expected to specify behavior more tightly than do standards for languages intended to maximize performance on every platform.

Because this standard requires facilities that are not currently available in common programming languages, the standards for such languages might not be able to fully conform to this standard if they are no longer being revised. If the language can be extended by a function library or class or package to provide a conforming environment, then that extension should define all the language-defined behaviors that would normally be defined by a language standard.

Implementation-defined behavior is defined by a specific implementation of a specific programming environment conforming to this standard. Implementations define behaviors not specified by this standard nor by any relevant programming language standard or programming language extension. Conformance to this standard is a property of a specific implementation of a specific programming environment, rather than of a language specification. However a language standard could also be said to conform to this standard if it were constructed so that every conforming implementation of that language also conformed automatically to this standard.

1.8. Language considerations. All relevant languages are based on the concepts of data and transformations. In Von Neumann languages, data are held in variables, which are transformed by assignment. In functional languages, input data are supplied as arguments; the transformed form is returned as results. Dataflow languages vary considerably, but use some form of the data and transformation approach.

Similarly, all relevant languages are based on the concept of mapping the pseudo-mathematical notation that is the program code to approximate real arithmetic, nowadays almost exclusively using some form of floating-point. The unit of mapping and transformation can be individual operations and built-in functions, expressions, statements, complete procedures, or other. This standard is applicable to all of these.

The least requirement on a conforming language standard, compiler or interpreter is that it shall:

(1) define bindings so that the programmer can specify level 2 data (in the sense of the levels defined in §**??**) as described in this standard;
(2) define bindings so that the programmer can specify the operations on such data as described in this standard;
(3) define any properties of such data and operations that this standard requires to be defined;

(4) honor the rules of interval transformations on such data and operations as described in this standard; such units of transformation that the language standard, compiler or interpreter uses.

Specifically, if the data before and after the unit of transformation are regarded as sets of mathematical intervals, the transformed form of all combinations of the elements (the real values) represented by the prior set shall be a member of the posterior set.

If a conforming language standard supports reproducible interval arithmetic it shall also:

(5) Use the data bindings as specified in point (1) above for reproducible operations;
(6) Define bindings to the reproducible operations as described in this standard;
(7) Define any modes and constraints that the programmer needs to specify or obey in order to obtain reproducible results.

If a conforming language standard supports both non-reproducible and reproducible interval arithmetic it shall also:

(8) Permit a reproducible transformation unit to be used as a component in a non-reproducible program, possibly via a suitable wrapping interface.

2. Normative references

1. IEEE Std 754-2008. IEEE Standard for Floating-Point Arithmetic.

3. Conformance Clause

⚠ This should probably moved to an earlier place in the standard. It might be put before all the normative things, which are referenced thereafter. Thus with normative notations it would be Chapter 1 Section 3.

To conform to OASIS Conformance Requirements for Specifications version 0.5, 1 March 2002 we have to do the following things in essence:

- ✓ provide a conformance clause—we are doing this right now
- ✓ use the proper conformance keywords—as long as we are adhering to §1.5 this is covered.
- address all topics in section 8 of the OASIS requirements
  - ✓ (8.1) Specify what conforms—an implementation of an interval arithmetic programming environment
  - ✓ (8.1.1) Modularity of the conformance—We don't have different components that individually have to conform.
  - ○ (8.1.2) Specifying conformance claims—We don't have different levels of conformance
  - – (8.2) Profiles and levels—the flavors look like profiles
  - – (8.3) Extensions—probably none
  - – (8.4) Discretionary items—implementation defined things
  - – (8.5) Deprecated things—not yet
  - – (8.6) Internationalization—english is normative, we have some characteristics for languages which might be interesting here
- examine and if appropriate address all things in section 9
  - – (9.1) Implementation Conformance Statement—this might be relevant to register implementation defined points
  - – (9.2) Test assertions—probable connection to the test suite
  - – (9.3) Testing methodology/program

Things to consider

- a language definition may be supporting the standard, an implementation may be conforming - may we have different objects that may be the object of the standard after all, see §1.7
- an implementation might also be a library, class or package
- a reproducibility mode might look like a level? at least it is a kind of quality of the implementation
- perhaps we have to clarify (here?) the difference between conformance levels and the levels structure—if necessary add something to the definitions.
- flavors
  - shall provide at least one
  - may provide additional
  - flavors following common evaluation on common intervals

An implementation of an interval arithmetic programming environment that conforms to this standard shall satisfy the following requirements

- flavor-independent requirements

A conforming implementation shall also provide at least one of the following profiles[1], defined in Clause 7, called included flavors of interval arithmetic in the context of this standard. In addition it may also provide additional flavors that

- shall provide the required operations in Clause 16;
- should provide the recommended operations in Clause 17;
- shall provide common evaluations on common intervals as specified in §7.4.

3.1. Set-based interval arithmetic. An implementation of the set-based flavor shall satisfy the following requirements

- set-based requirements

[1]maybe drop this term?

3.2. Kaucher interval arithmetic. An implementation of the Kaucher flavor shall satisfy the following requirements

- Kaucher requirements

An implementation may claim its conformance to this standard in the following way

> *Name of implementation and version* have been tested for conformance to the P1788 Standard for Interval Arithmetic, version D7.0 using the *we might put a reference to a probable test suite here* on *YYYY-MM-DD* and no nonconformities were found.

4. Notation, abbreviations, definitions

4.1. Frequently used notation and abbreviations.

754	IEEE-Std-754-2008 “IEEE Standard for Floating-Point Arithmetic”.
$\mathbb{R}$	the set of real numbers.
$\overline{\mathbb{R}}$	the set of extended real numbers, $\mathbb{R} \cup \{-\infty, +\infty\}$.
$\overline{\mathbb{IR}}$	the set of closed real intervals, including unbounded intervals and the empty set.
$\mathbb{F}, \mathbb{G}, \ldots$	generic notation for (the set of numbers, including $\pm\infty$) representable in some floating point format.
$\overline{\mathbb{I}}\mathbb{F}, \overline{\mathbb{I}}\mathbb{G}, \ldots$	the members of $\overline{\mathbb{IR}}$ whose lower and upper bounds are in $\mathbb{F}, \mathbb{G}, \ldots$.
Empty	the empty set.
Entire	the whole real line.
NaI	Not an Interval.
NaN	Not a Number.
qNaN	quiet NaN.
sNaN	signaling NaN.
$x, y, \ldots$ [resp. $f, g, \ldots$]	typeface/notation for a numeric value [resp. numeric function].
$\boldsymbol{x}, \boldsymbol{y}, \ldots$ [resp. $\boldsymbol{f}, \boldsymbol{g}, \ldots$]	typeface/notation for an interval value [resp. interval function].
$\mathsf{f}, \mathsf{g}, \ldots$	typeface/notation for an expression, producing a function by evaluation.
$\mathrm{Dom}(f)$	the domain of a point-function f.
$\mathrm{Rge}(f \mid \boldsymbol{s})$	the range of a point-function f over a set $\boldsymbol{s}$; the same as the image of $\boldsymbol{s}$ under f.

[*Note. Little used in this document, but used in classical interval analysis, are* $\mathbb{IR}$, *the set of bounded, nonempty closed real intervals; and* $\mathbb{IF}$, *the intervals of* $\mathbb{IR}$ *whose bounds are in* $\mathbb{F}$. *The symbols* $\overline{\mathbb{I}}$ *and* $\mathbb{I}$ *act as operators on subsets* S *of* $\overline{\mathbb{R}}$, *namely* $\overline{\mathbb{I}}S$ *or* $\overline{\mathbb{I}}(S)$ = *“the empty set, plus all intervals whose endpoints are in* S*” and* $\mathbb{I}S$ = *“all nonempty intervals whose endpoints are finite and in* S*”.*]

4.2. Definitions.

⚠ Definitions belonging to Levels 2 onward have been temporarily removed.

4.2.1. **arithmetic operation.** A function provided by an implementation (see Defn 4.2.8). It comes in three forms: the **point** operation, which is a mathematical real function of real variables such as $\log(x)$; one or more **(bare) interval extensions** of the point operation, each of which corresponds to the finite precision interval type of its result; and one or more **decorated interval extensions**, each being the (unique) decorated version of a bare interval extension.

Together with the interval non-arithmetic operations (§9.4.1), these form the implementation's **library**, which splits into the **point library** (a conceptual entity, being a set of mathematical functions), the **bare interval library** and the **decorated interval library**, corresponding to the above categories. The latter two may be further qualified by a result interval type, e.g., “binary64 inf-sup decorated interval library”.

The programming environment's floating point approximations to mathematical point functions constitute the *floating point library.* The standard makes no requirements on these.

A **basic arithmetic operation** is one of the six functions $+$, $-$, $\times$, $\div$, fused multiply-add `fma` and square root `sqrt`.

Constants such as 3.456 and π are regarded as arithmetic operations whose number of arguments is zero. Details in §9.4.4.

4.2.2. **box.** See Defn 4.2.10.

4.2.3. **domain.** For a function with arguments taken from some set, the **domain** comprises those points in the set at which the function has a value. The domain of an arithmetic operation is part of its definition. E.g., the (point) arithmetic operation of division x/y, in this standard, has arguments (x, y) in $\mathbb{R}^2$, and its domain is the set $\{\,(x, y) \in \mathbb{R}^2 \mid y \neq 0\,\}$. See also Defn 4.2.14.

4.2.4. **elementary function.** Synonymous with arithmetic operation.

4.2.5. **expression.** A symbolic object f that is either a symbolic variable or, recursively, of the form $\varphi(\mathsf{g}_1, \ldots, \mathsf{g}_k)$, where φ is the name of a k-argument arithmetic or non-arithmetic operation and the g_i are expressions. It is an **arithmetic expression** if all its operations are arithmetic operations. Writing $f(z_1, \ldots, z_n)$ makes f a **bound expression**, giving it an **argument list** comprising the variables z_i in that order, which must include all those that occur in f.

Details in §9.5. For the ways in which an expression defines a function, see §**??**, **??**.

4.2.6. **fma.** Fused multiply-add operation, that computes $x \times y + z$. One of the basic arithmetic operations.

4.2.7. **hull.** (Full name: **interval hull**.) When not qualified by the name of a finite-precision interval type, the hull of a subset $\boldsymbol{s}$ of $\mathbb{R}$ is the tightest (q.v.) interval containing $\boldsymbol{s}$.

4.2.8. **implementation.** When used without qualification, means a realization of an interval arithmetic conforming to the specification of this standard.

4.2.9. **inf-sup.** Describes a representation of an interval based on its lower and upper bounds.

4.2.10. **interval.** A closed connected subset of $\mathbb{R}$; may be empty, bounded or unbounded. May be called a 1-dimensional interval, see next paragraph. The set of all intervals is denoted $\overline{\mathbb{IR}}$.

A **box** or **interval vector** is an n-dimensional interval, i. e. a tuple $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$ where the $\boldsymbol{x}_i$ are intervals. Often identified with the cartesian product $\boldsymbol{x}_1 \times \ldots \times \boldsymbol{x}_n \subseteq \mathbb{R}^n$, it is empty if any of the $\boldsymbol{x}_i$ is empty. Details in §9.2.

4.2.11. **interval extension.** An interval extension of a point function f is a function $\boldsymbol{f}$ from intervals to intervals such that $f(x)$ belongs to $\boldsymbol{f}(\boldsymbol{x})$ whenever x belongs to $\boldsymbol{x}$ and $f(x)$ is defined. Details in §9.4.3. It is the **natural** (or tightest) interval extension if $\boldsymbol{f}(\boldsymbol{x})$ is the interval hull of the range of f over $\boldsymbol{x}$, for all $\boldsymbol{x}$.

4.2.12. **interval function, interval mapping.** A function from intervals to intervals is called an interval function if it is an interval extension of a point function, and an interval mapping otherwise. Details in §9.4.3.

4.2.13. **interval library.** See Defn 4.2.1.

4.2.14. **natural domain.** For an arithmetic expression $f(z_1, \ldots, z_n)$, the natural domain is the set of $x = (x_1, \ldots, x_n) \in \mathbb{R}^n$ where the expression defines a value for the associated point function $f(x)$. See §**??**.

4.2.15. **no value vs. undefined.** Some functions do not have a defined value at the mathematical model of Level 1 (see Clause 5). This translates to a defined value at the interval datum Level 2 given at the corresponding places in the standard. Therefore the term “no value” is not to be confused with an “undefined” value, which has the common meaning of a value not determined by the standard and thus being free for the implementation to decide.

4.2.16. **non-arithmetic operation.** An operation on intervals that is not an interval extension of a point operation; includes interval intersection and union.

4.2.17. **number.** Any member of the set $\mathbb{R} \cup \{-\infty, +\infty\}$ of extended reals: a **finite number** if it belongs to $\mathbb{R}$, else an **infinite number**. See §9.1.

4.2.18. **point function, point operation.** A mathematical function of real variables: that is, a map f from its domain, which is a subset of $\mathbb{R}^n$, to $\mathbb{R}^m$, where $n \geq 0, m > 0$. It is **scalar** if $m = 1$. Any arithmetic expression $f(z_1, \ldots, z_n)$ defines a (usually scalar) point function, whose domain is the natural domain of f.

4.2.19. **point library.** See Defn 4.2.1.

4.2.20. **range.** The range, $\mathrm{Rge}(f \mid \boldsymbol{s})$, of a point function f over a subset $\boldsymbol{s}$ of $\mathbb{R}^n$ is the set of all values that f assumes at those points of $\boldsymbol{s}$ where it is defined, i.e. $\{\, f(x) \mid x \in \boldsymbol{s} \text{ and } x \in \mathrm{Dom}\, f \,\}$.

4.2.21. **string.** A text string, or just string, is a finite sequence of characters belonging to some alphabet. See §9.1.

4.2.22. **tightest.** Smallest in the partial order of set containment. The tightest set (unique, if it exists) with a given property is contained in every other set with that property.

4.2.23. **version.** For brevity, bare or decorated interval extensions, or floating point approximations, of a library point operation may be called versions of that operation.

Relationships between specification levels for interval arithmetic for a given flavor $\mathfrak{F}$ and a given finite-precision interval type $\mathbb{T}$.		
Level 1	Number system used by flavor $\mathfrak{F}$. Set of allowed intervals in $\mathfrak{F}$. Principles of how $+$, $-$, $\times$, $\div$ and other arithmetic operations are extended to intervals.	Mathematical Model level.
	$\downarrow$ $\mathbb{T}$-*interval hull* total, many-to-one[a] — *identity map* $\uparrow$ total, one-to-one[b]	
Level 2	A finite subset $\mathbb{T}$ of the $\mathfrak{F}$-intervals—the $\mathbb{T}$-interval datums—and operations on them.	Interval datum level.
	"represents" $\uparrow$ partial, many-to-one, onto[c]	
Level 3	Representations of $\mathbb{T}$-intervals, e.g. by two floating point numbers.	Representation level.
	"encodes" $\uparrow$ partial, many-to-one, onto[d]	
Level 4	Encodings `0111000...`	Bit string level.

TABLE 1. Specification levels for interval arithmetic

5. Structure of the standard in levels

The standard is structured into four levels, summarized in Table 1, that match the levels defined in the 754 standard, see 754 Table 3.1.

Level 1, in Clause 9, defines the mathematical theory underlying the standard. The entities at this level are mathematical intervals and operations on them. Conforming implementations shall implement this theory.

In addition to an ordinary (bare) interval, this level defines a *decorated* interval, comprising a bare interval and a *decoration*. Decorations implement this standard's exception handling mechanism.

Level 2, in Clause 11, is the central part of the standard. Here the mathematical theory is approximated by an implementation-defined finite set of entities and operations. A level 2 entity is called a *datum* (plural "datums" in this standard, since "data" is often misleading).

An interval datum is a mathematical interval tagged by a unique *type* $\mathbb{T}$. The type abstracts a *representation scheme*—a particular way of representing intervals (e.g., by storing their lower and upper bounds as IEEE `binary64` numbers). Level 2 arithmetic normally acts on intervals of a given type to produce an interval of the same type (but interval operations that act on intervals of types other than the result type are possible).

Level 3, in Clause 14, is concerned with the representation of interval datums—usually but not necessarily in terms of floating point values. A level 3 entity is an *interval object*. Representations of decorations, hence of decorated intervals, are also defined at this level.

The Level 3 requirements in this standard are few, and concern mappings from internal representations to external ones, such as interchange formats and I/O.

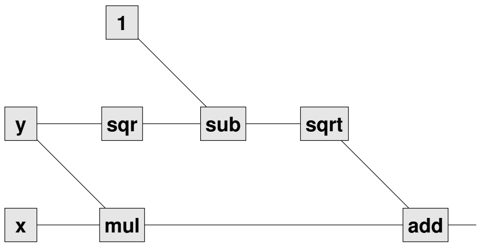
A level 4 entity is a *bit string*. This standard makes no Level 4 requirements.

The arrows in Table 1 denote mappings between levels. The phrases in italics name these mappings. Each phrase "total, many-to-one", etc., labeled with a letter [a] to [d], is descriptive of the mapping and is equivalent to the corresponding labeled fact below.

a. Each mathematical interval (indeed each subset of $\mathbb{R}$) has a unique interval datum as its $\mathbb{T}$-*hull*—a minimal enclosing interval of the type $\mathbb{T}$.
b. Each interval datum is a mathematical interval, if one ignores its type-tag.
c. Not every interval object necessarily represents an interval datum, but when it does, that datum is unique. Each interval datum has at least one representation, and may have more than one.
d. Not every interval encoding necessarily encodes an interval object, but when it does, that object is unique. Each interval object has at least one encoding and may have more than one.

$\sqrt{y^2-1}+xy$

```
input  $x, y$
$v_1 = y^2$
$v_2 = v_1 - 1$
$v_3 = \sqrt{v_2}$
$v_4 = x \times y$
$z = v_3 + v_4$
output  $z$
```



Formula (a) | Pseudo-code (b) | Computational graph (c)

FIGURE 1. Essentially equivalent notations for an expression. The pseudo-code (b) breaks the formula into single library operations, and uses static single-assignment (SSA) form. The graph form (c) uses the names in §9.6.3, for library operations.

6. Expressions and the functions they define

An expression is some symbolic form used to define a function. Expressions are central to interval computation, because the Fundamental Theorem of Interval Arithmetic (FTIA) is about interpreting an expression in different ways: first, as defining a mathematical real point function f; second, as defining interval mappings that give proven enclosures for the range of f over an input box $\boldsymbol{x}$. The decoration system, in suitable situations, lets one strengthen this by concluding f is everywhere defined, or everywhere continuous, on $\boldsymbol{x}$— making it possible, for example, to automatically verify the assumptions of the Brouwer Fixed Point Theorem.

The standard specifies behavior at the individual operation level, that makes such conclusions possible. It does not define a meaning of "expression" since this would raise language-dependent semantic issues outside its scope. However it is useful to list some things that an expression is and is not, to give a context for the specifications of the standard:

(i) An expression f defines a relation between certain *inputs*, often represented by mathematical or programming variables, and certain *outputs*, via the application of named *operations* that come from a *library*. f may be represented in various ways, e.g., by a normal algebraic formula or by a segment of program code or pseudo-code or as a computational graph. This is illustrated for (an expression defining) the function $f(x, y) = \sqrt{y^2-1}+xy$ in Figure 1 (a,b,c) respectively.

Code may define a *vector* expression, with several outputs; however, all the individual library operations of the standard are *scalar*, with possibly several inputs but a single output.

(ii) For the FTIA to apply, f must be an *arithmetic expression*, each of whose library operations is primarily a *point-operation* with real-number input(s) and output. The operations must be *generic*, each one having one or more *interval version(s)* with interval input and output, and corresponding *decorated interval version(s)*. These produce *point evaluation*, *interval evaluation* and *decorated interval evaluation* of the expression, also termed evaluation in *point mode*, *interval mode* or *decorated interval mode*.

An implementation's library by definition comprises all its finite-precision (Level 2) versions of required or recommended operations that it provides for any of its supported interval types, as specified in §9.6, §9.7 and in Clause 11. Different Level 2 interval evaluations of f come from using library operations with different combinations of input and output types (precisions), as the implementation may provide.

The set operations `intersection` and `convexHull` are not point-operations and cannot appear in an arithmetic expression. However they are useful for efficiently *implementing* interval versions of functions defined piecewise (see Example (ii) in §10.10).

An arithmetic operation or expression may also denote a floating point function; these are not specified by this standard.

(iii) The expression must be *explicit*. Interval analysis has ways to find range-enclosures for a function such as "$y = g(x)$ defined implicitly by solving $\sqrt{y^2 - 1} + xy = 1$ for y"; but the FTIA does not apply directly to such functions.

Each library point-operation has a defined domain, the set of inputs where it can be evaluated. This leads to the idea of *natural domain* $\operatorname{Dom}(f)$ of the point function $f(x) = f(x_1, \ldots, x_n)$ defined by an expression: the set of points x where f is *defined* in the sense that the whole expression can be successfully evaluated. [*Example. From the domains of* / *and* $\sqrt{\cdot}$, *one finds the natural domain of* $\sqrt{1 + 1/x}$ *is the union of the two intervals* $(-\infty, -1]$ *and* $(0, +\infty)$.]

In the set-based flavor—see §7.5 for other flavors—Moore's basic theorem for a scalar function is as follows, with the above notation.

Theorem 6.1 (Fundamental Theorem of Interval Arithmetic)**.** *Let* $\boldsymbol{y} = f(\boldsymbol{x})$ *be the result of interval-evaluation of* f *over a box* $\boldsymbol{x} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$ *using any interval versions of its component library functions. Then*

(i) ("Weak" form of FTIA.) In all cases, $\boldsymbol{y}$ *contains the range of* f *over* $\boldsymbol{x}$*, that is, the set of* $f(x)$ *at points of* $\boldsymbol{x}$ *where it is defined:*

$$\boldsymbol{y} \supseteq \operatorname{Rge}(f \mid \boldsymbol{x}) = \{\, f(x) \mid x \in \boldsymbol{x} \cap \operatorname{Dom}(f) \,\}. \tag{1}$$

(ii) ("Defined" form of FTIA.) If also each library operation in f *is everywhere defined on its inputs, while evaluating* $\boldsymbol{y}$*, then* f *is everywhere defined on* $\boldsymbol{x}$*, that is* $\operatorname{Dom}(f) \supseteq \boldsymbol{x}$*.*

(iii) ("Continuous" form of FTIA.) If in addition to (ii), each library operation in f *is everywhere continuous on its inputs, while evaluating* $\boldsymbol{y}$*, then* f *is everywhere continuous on* $\boldsymbol{x}$*.*

It is important to note that the theorem holds in finite precision (Level 2), not just at Level 1. Each of its forms is useful; e.g., the "continuous" FTIA is needed in applications where the assumptions for applying Brouwer's Fixed Point Theorem must be verified. Building on features in older interval systems, the standard's decoration system, Clause 8, gives basic tools for checking the conditions for the "defined" and "continuous" forms, during evaluation of a function.

The following points are also relevant; they concern language semantics of expressions.

(iv) The computational graph of the expression is a directed acyclic graph. When program code is involved, the FTIA applies to the expression f evaluated in a particular execution of that code. If code contains conditional statements or loops, f and its graph may change from one execution to another. Often such code is designed to define a function piecewise (e.g., over several intervals, see example in §10.10). The user is responsible for checking that the FTIA applies as intended in such cases; the standard provides no way to check automatically that a property such as continuity holds globally.

(v) There is a mismatch between expressions and set theory when doing interval-evaluation, illustrated by the following. Define a 3-argument function by

$$f(x, y, z) = x + z, \tag{2}$$

where argument y does not occur in the expression. Interval-evaluation $\boldsymbol{w} = f(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})$ returns $\boldsymbol{x} + \boldsymbol{z}$, ignoring $\boldsymbol{y}$. When $\boldsymbol{y}$ is empty, e.g.

$$\boldsymbol{w} = f([1, 2], \emptyset, [3, 4]) = [4, 6], \tag{3}$$

this is at odds with set-theory, which says $\boldsymbol{w}$ encloses the range of the point-function over the product box $[1, 2] \times \emptyset \times [3, 4]$. Since this is empty, the range is empty, suggesting the computed $\boldsymbol{w}$ should be $\emptyset$ instead of $[4, 6]$.

It is language-defined which of these interpretations is adopted, but the FTIA theory in Annex D is framed so that the theoretically correct result coincides with what straightforward interval evaluation computes—here $[4, 6]$. This is done by defining the function specified by an expression to depend only on those inputs that in the evaluation's computational graph *lie on a path to the output*. In (2), the function thus has two arguments x, z instead of three.

This is conveniently handled by using *keyword association* of dummy to actual arguments—for expressions, not for individual library operations for which normal positional arguments are used—so that irrelevant variables can be skipped. E.g. the argument association in (3) may be written $f(x = [1, 2],\ z = [3, 4])$. See Annex D for details.

7. Flavors

7.1. Flavors overview. The standard permits different interval *flavors*, which embody different foundational (Level 1) approaches to intervals. An implementation shall provide at least one flavor. For brevity, phrases such as "A flavor shall provide, or document, a feature" mean that the implementation of that flavor shall provide the feature, or its documentation describe it.

Flavor is a property of program execution context, not of an individual interval, therefore just one flavor shall be in force at any point of execution. It is recommended that at the language level, the flavor should be constant at the level of a procedure/function, or of a compilation unit.

A flavor is identified by a unique name. Certain flavors, termed **included**, are specified in this standard. The *(list to be confirmed)* flavors are the currently included flavors. The procedure for submitting a new flavor for inclusion is described in Annex B. A conforming implementation that provides one or more included flavors may also provide non-included flavors, without losing conformance for the included flavors.

The flavor concept enforces a common core of behavior that different kinds of interval arithmetic must share:

(i) The set of required operations, identified by their names, is the same in all flavors. Similarly the set of recommended operations is the same in all flavors. See Clause 16, Clause 17.
(ii) There is a set of *common intervals* whose members are—in a sense made precise below—intervals of any flavor.
(iii) There is a set of common evaluations of library operations, with common intervals as input, that give—again in a sense made precise below—the same result in any flavor.

The result in item (iii) is a mathematically tightest (Level 1) result, ignoring any interval widening due to finite precision (Level 2).

7.2. Definition of common intervals and common evaluations. The choice of the set of common intervals, and the set of common evaluations of an operation, is a design decision that defines the flavor concept. It should aim for simplicity, and the common evaluations should be specified by a general rule that makes it easy to add a new operation to the library if needed. The choice that was made is specified in the following paragraphs.

All likely flavors extend the classical Moore arithmetic [**5**] on the set $\mathbb{IR}$ of *closed bounded nonempty real intervals*, and there is no interval outside $\mathbb{IR}$ that is supported in all these flavors. Hence, the chosen set $\mathfrak{C}$ of common intervals is $\mathbb{IR}$.

The common evaluations are specified in terms of graphs of interval operations. For an interval operation φ of arity k, its graph (in some flavor) is a subset of a $(k{+}1)$-dimensional space of intervals, namely the set of interval $(k{+}1)$-tuples $(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_k;\ \boldsymbol{y})$ such that $\varphi(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_k) = \boldsymbol{y}$ is true in that flavor. Each such tuple is called an *operation instance.*

The general rule is that each φ has a set $\mathfrak{CE}(\varphi)$ of **common evaluations**: operation instances $(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_k;\ \boldsymbol{y})$ such that all its components are in $\mathbb{IR}$ and

$$\varphi(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_k) = \boldsymbol{y} \qquad \text{shall hold in all flavors.}$$

$\mathfrak{CE}(\varphi)$ may be regarded as the *flavor-independent graph* of φ. For brevity, writing

$$\text{the evaluation } \varphi(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_k) = \boldsymbol{y} \text{ is common,} \tag{4}$$

or the equivalent notation when φ is an infix operator (e.g., $\boldsymbol{x}_1 + \boldsymbol{x}_2 = \boldsymbol{y}$), means that $(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_k;\ \boldsymbol{y}) \in \mathfrak{CE}(\varphi)$.

The standard defines $\mathfrak{CE}(\varphi)$ as follows.

Arithmetic operation: Such an operation is an interval extension, in fact the natural interval extension (Defn 4.2.11), of the corresponding point function φ. The common operation instances are those $(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_k;\ \boldsymbol{y})$ such that the point function φ is *defined and continuous at each point of* the closed, bounded, nonempty box $\boldsymbol{x} = (\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_k)$, and $\boldsymbol{y}$ equals the range of φ over this box. Then necessarily $\boldsymbol{y}$ belongs to $\mathbb{IR}$.

Non-arithmetic operation: The common operation instances are those tuples with common inputs $\boldsymbol{x}_i$ such that *the result $\boldsymbol{y}$ is also common.* In particular for `convexHull`, the common operation instances are those $(\boldsymbol{x}_1, \boldsymbol{x}_2;\ \boldsymbol{y})$ with arbitrary $\boldsymbol{x}_1, \boldsymbol{x}_2 \in \mathbb{IR}$ and $\boldsymbol{y}$ equal to the convex hull of $\boldsymbol{x}_1 \cup \boldsymbol{x}_2$. For `intersection`, they are those $(\boldsymbol{x}_1, \boldsymbol{x}_2;\ \boldsymbol{y})$ with arbitrary $\boldsymbol{x}_1, \boldsymbol{x}_2 \in \overline{\mathbb{IR}}$ and $\boldsymbol{y}$ equal to $\boldsymbol{x}_1 \cap \boldsymbol{x}_2$, provided the latter is *nonempty* (since $\emptyset \notin \overline{\mathbb{IR}}$).

[*Examples.*

- *The evaluation* $[-1,4]/[3,4] = [-1/3, 4/3]$ *is common; but* $[3,4]/[-1,4] = \boldsymbol{y}$ *is not common, for any* $\boldsymbol{y} \in \overline{\mathbb{IR}}$.
- *In the set-based flavor,* $\sqrt{[-1,4]} = [0,2]$. *But in the Kaucher flavor* $\sqrt{[-1,4]}$ *is undefined, so* $\sqrt{[-1,4]} = \boldsymbol{y}$ *cannot be common for any* $\boldsymbol{y}$. *In fact* $\mathfrak{CE}(\texttt{sqrt})$ *is the set of all* $([a,b];\ [\sqrt{a}, \sqrt{(b)}])$ *for which* $0 \le a \le b < +\infty$.
- *Similarly,* $([-1,4] \cap [5,6] = \emptyset)$ *in the set-based flavor, while* $([-1,4] \cap [5,6] = [5,4])$ *in the Kaucher flavor. Thus* $([-1,4] \cap [5,6] = \boldsymbol{y})$ *cannot be common for any* $\boldsymbol{y}$.
- *The above definition for arithmetic operations requires R1 "φ is defined and continuous at each point of $\boldsymbol{x}$", which is a* stronger *constraint on* $\mathfrak{CE}(\varphi)$ *(results in fewer common evaluations) than R2 "the restriction of φ to $\boldsymbol{x}$ is everywhere defined and continuous". This produces a* weaker *constraint on what interval arithmetics can be flavors (with fewer rules, more arithmetics obey them). In particular, requirement R1 permits cset arithmetic to be a flavor, while R2 does not.*

  E.g., the common evaluations of the function $\mathsf{floor}(x)$ *are all* $([a,b];\ [k,k])$ *with* $k \in \mathbb{Z}$, $k < a \le b < k+1$. *Thus* $\mathsf{floor}([1,1.9]) = [1,1]$ *is not common, because* $\mathsf{floor}()$ *is not continuous at* 1, *despite its restriction to* $[1,1.9]$ *being everywhere continuous. If it were required to be common, cset arithmetic could not be a flavor.*

]

7.3. Loose common evaluations. At Level 2, common evaluations are usually not computable because of roundoff; instead, an enclosing interval of some finite precision interval type is computed. The notion of a **loose common evaluation** takes account of this. A loose common evaluation derived from a common evaluation $\varphi(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_k) = \boldsymbol{y}$ in $\mathfrak{CE}(\varphi)$ is defined to be any

$$\varphi(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_k) = \boldsymbol{y}' \tag{5}$$

where $\boldsymbol{y}'$ is a member of $\overline{\mathbb{IR}}$ containing $\boldsymbol{y}$. An element of $\mathfrak{CE}(\varphi)$ is itself loose by this definition, but may be called a **tight** common evaluation to emphasize that its $\boldsymbol{y}$ is contained in each derived $\boldsymbol{y}'$, above.

Informally, for a given φ and $\boldsymbol{x} = (\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_k)$, the loose common evaluations describe all closed bounded intervals that might be produced by evaluating an enclosure of $\operatorname{Rge}(\varphi \,|\, \boldsymbol{x})$ in finite precision.

7.4. Relation of common evaluations to flavors. The formal definition of common evaluations takes into account that the common intervals are not necessarily a subset of the intervals of a given flavor, but are identified with a subset of it by an embedding map.

[*Examples.*

A Kaucher interval is defined to be a pair (a,b) *of real numbers—equivalently, a point in the plane* $\mathbb{R}^2$*—which for* $a \le b$ *is "proper" and identified with the normal real interval [a,b], and for* $a > b$ *is "improper". Thus the embedding map is* $\boldsymbol{x} \mapsto (\inf \boldsymbol{x}, \sup \boldsymbol{x})$ *for* $\boldsymbol{x} \in \overline{\mathbb{IR}}$.

For the set-based flavor, every common interval is actually an interval of that flavor ($\overline{\mathbb{IR}}$ *is a subset of* $\overline{\mathbb{IR}}$*), so the embedding is the identity map* $\boldsymbol{x} \mapsto \boldsymbol{x}$ *for* $\boldsymbol{x} \in \overline{\mathbb{IR}}$.]

Formally, a flavor is identified by a pair $(\mathfrak{F}, \mathfrak{f})$ where $\mathfrak{F}$ is a set of Level 1 entities, the *intervals* of that flavor, and $\mathfrak{f}$ is a one-to-one *embedding map* $\overline{\mathbb{IR}} \to \mathfrak{F}$. Usually, $\mathfrak{f}(\boldsymbol{x})$ is abbreviated to $\mathfrak{f}\boldsymbol{x}$.

It is then required that *operation compatibility* shall hold for each library operation φ and for each flavor $(\mathfrak{F}, \mathfrak{f})$. Namely, given $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_k$ and $\boldsymbol{y}$ in $\overline{\mathbb{IR}}$,

If $(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_k;\ \boldsymbol{y})$ is a common operation instance of φ,
then $(\mathfrak{f}\boldsymbol{x}_1, \mathfrak{f}\boldsymbol{x}_2, \ldots, \mathfrak{f}\boldsymbol{x}_k;\ \mathfrak{f}\boldsymbol{y})$ is an operation instance of φ in flavor $\mathfrak{F}$. (6)

That is, if the evaluation $\varphi(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_k) = \boldsymbol{y}$ is common, then $\varphi(\mathfrak{f}\boldsymbol{x}_1, \mathfrak{f}\boldsymbol{x}_2, \ldots, \mathfrak{f}\boldsymbol{x}_k)$ must be defined in $\mathfrak{F}$ with value $\mathfrak{f}\boldsymbol{y}$.

An evaluation in $\mathfrak{F}$ of an expression, in which only (loose) common evaluations of elementary operations occur, is called a common evaluation of that expression. That is, in a flavor $(\mathfrak{F}, \mathfrak{f})$, the expression's inputs are members of $\mathfrak{f}(\overline{\mathbb{IR}})$, and each intermediate value is produced by a common evaluation of an operation so that it is also in $\mathfrak{f}(\overline{\mathbb{IR}})$; hence the final result is in $\mathfrak{f}(\overline{\mathbb{IR}})$.

The `com` decoration makes it possible to determine, for a specific expression and specific interval inputs, whether common evaluation has occurred, see Clause 8.

7.5. Flavors and the Fundamental Theorem. For a common evaluation of an arithmetic expression, each library operation is (i.e., can be regarded as, modulo the embedding map) defined and continuous on its inputs so that it satisfies the conditions of the strongest, "continuous" form of the FTIA, Theorem 6.1. At Level 1, using the tightest interval extension of each operation, the range enclosure obtained by a common evaluation is (again modulo the embedding map) the same, independent of flavor.

It is possible in principle for an implementation to make this true also at Level 2, by providing shared number formats and interval types that represent the same sets of reals or intervals in each flavor; and library operations on these types and formats that have identical numerical behavior in each flavor. For example, both set-based and Kaucher flavors might use intervals stored as two IEEE754 `binary64` numbers representing the lower and upper bounds, and might ensure that operations, when applied to the intervals recognized by both flavors, behave identically. Such shared behavior might be useful for testing correctness of an implementation.

Beyond common evaluations, versions of the FTIA in different flavors can be strictly incomparable. For example, the set-based FTIA handles unbounded intervals, which the Kaucher flavor does not; while Kaucher intervals have an extended FTIA, involving generalized meanings of "contains" and "interval extension" applicable to reverse-bound intervals, which has no simple interpretation in the set-based flavor.

8. Decoration system

8.1. Decorations overview. A decoration is information attached to an interval; the combination is called a decorated interval. Interval calculation has two main objectives:

– obtaining correct range enclosures for real-valued functions of real variables;
– verifying the assumptions of existence, uniqueness, or nonexistence theorems.

Traditional interval analysis targets the first objective; the decoration system, as defined in this standard, targets the second.

A decoration primarily describes a property, not of the interval it is attached to, but of the function defined by some code that produced the interval by evaluating over some input box.

For instance, if a section of code defines the expression $\sqrt{y^2 - 1} + xy$, then decorated-interval evaluation of this code with suitably initialized input intervals $\boldsymbol{x}, \boldsymbol{y}$ gives information about the definedness, continuity, etc. of the point function $f(x, y) = \sqrt{y^2 - 1} + xy$ over the box $(\boldsymbol{x}, \boldsymbol{y})$ in the plane.

The decoration system is designed in a way that naive users of interval arithmetic do not notice anything about decorations, unless they inquire explicitly about their values. They only need

– call the `newDec` operation on the inputs of any function evaluation used to invoke an existence theorem,
– explicitly convert relevant floating-point constants (but not integer parameters such as the p in $\texttt{pown}(x, p) = x^p$) to intervals,

and have the full rigor of interval calculations available. A smart implementation may even relieve users from these tasks. Expert users can inspect, set and modify decorations to improve code efficiency, but are responsible for checking that computations done in this way remain rigorously valid.

Especially in the set-based flavor, decorations are based on the desire that, from an interval evaluation of a real function f on a box $\boldsymbol{x}$, one should get not only a range enclosure $f(\boldsymbol{x})$ but also a guarantee that the pair $(f, \boldsymbol{x})$ has certain important properties, such as $f(x)$ being defined for all $x \in \boldsymbol{x}$, f restricted to $\boldsymbol{x}$ being continuous, etc. This goal is achieved, in parts of a program that require it, by performing *decorated interval evaluation*, whose semantics is summarized as follows:

Each intermediate step of the original computation depends on some or all of the inputs, so it can be viewed as an intermediate function of these inputs. The result interval obtained on each intermediate step is an enclosure for the range of the corresponding intermediate function. The decoration attached to this intermediate interval reflects the available knowledge about whether this *intermediate function is guaranteed to be everywhere defined, continuous, bounded, etc., on the given inputs.*

In some flavors, certain interval operations ignore decorations, i.e., give undecorated interval output. Users are responsible for the appropriate propagation of decorations by these operations.

The function f is assumed to be expressed by code, an algebraic formula, etc.—generically termed an *expression*—which can be evaluated in several modes: point evaluation, interval evaluation, or decorated interval evaluation. The standard does not specify a definition of "expression"; however, Annex D gives formal proofs in terms of a particular definition, and indicates how this relates to expressions in some programming languages.

This standard's decoration model, in contrast with 754's, has no status flags. A general aim, as in 754's use of NaN and flags, is not to interrupt the flow of computation: rather, to collate information while evaluating f, that can be inspected afterwards. This enables a fully local handling of exceptional conditions in interval calculations—important in a concurrent computing environment.

An implementation may provide any of the following: (i) status flags that are raised in the event of certain decoration values being produced by an operation; (ii) means for the user to specify that such an event signals an exception, and to invoke a system- or user-defined handler as a result. [*Example. The user may be able to specify execution be terminated if an arithmetic operation is evaluated on a box that is not wholly inside its domain—an interval version of 754's "invalid operation" exception.*] Such features are language- or implementation-defined.

8.2. Decoration definition and propagation. Each flavor shall document its set of provided decorations and their mathematical definitions. These are flavor-defined, with the exception of the decoration `com`, see §8.3.

The implementation makes the decoration system of each flavor available to the user via *decorated interval extensions* of relevant library operations. Such an operation φ, with interval inputs $\boldsymbol{x}_1, \dots, \boldsymbol{x}_k$ carrying decorations $dx_1, \dots, dx_k$, shall compute the same interval output $\boldsymbol{y}$ as the corresponding bare interval extension of φ—hence dependent on the $\boldsymbol{x}_i$ but not on the dx_i. It shall compute a *local decoration* d, dependent on the $\boldsymbol{x}_i$ and possibly on $\boldsymbol{y}$, but not on the dx_i. It shall combine d with the dx_i by a flavor-defined *propagation rule* to give an output decoration dy, and return $\boldsymbol{y}$ decorated by dy.

The local decoration d may convey purely Level 1 information—e.g., that φ is everywhere continuous on the box $\boldsymbol{x} = (\boldsymbol{x}_1, \dots, \boldsymbol{x}_k)$. It may convey Level 2 information related to the particular finite-precision interval types being used—e.g., that $\boldsymbol{y}$, though mathematically a bounded interval, became unbounded by overflow. For diagnostic use it may convey Level 3 or 4 information, e.g., how an interval is represented, or how memory is used.

If f is an expression, decorated interval evaluation of an expression means evaluation of f with decorated interval inputs and using decorated interval extensions of the expression's library operations. Those inputs generally need to be given suitable initial decorations that lead to the most informative output-decoration. A flavor shall provide a `newDec` function for this purpose. If $\boldsymbol{x}$ is a bare interval, `newDec`$(\boldsymbol{x})$ equals $\boldsymbol{x}$ with such an initial decoration.

It is the responsibility of each flavor to document the meaning of its decorations, and the correct use of these decorations within programs.

8.3. Recognizing common evaluation. A flavor may provide the decoration `com` with the following propagation rule for library arithmetic operations. In an implementation with more than one flavor, each flavor shall do so.

In the following, φ denotes an arbitrary interval extension of a point library arithmetic operation φ, provided by the implementation at Level 2 (typically the one associated with a particular interval type).

> Let φ applied to input intervals $\boldsymbol{x}_1, \boldsymbol{x}_2, \dots, \boldsymbol{x}_k$ give the computed result $\boldsymbol{y}$, and let $\varphi(\boldsymbol{x}_1, \boldsymbol{x}_2, \dots, \boldsymbol{x}_k) = \boldsymbol{y}$ be a loose common evaluation as defined in (5); in particular $\boldsymbol{y}$ is bounded. If each of the inputs $\boldsymbol{x}_i$ is decorated `com`, then the output $\boldsymbol{y}$ shall be decorated `com`.

Informally, `com` records that the individual operation φ took bounded nonempty input intervals and produced a bounded (necessarily nonempty) output interval. This can be interpreted as indicating "overflow did not occur". Further, the propagation rule ensures that if the initial inputs to an arithmetic expression f are bounded and nonempty, and are initialized with the decoration `com`, then the final result $\boldsymbol{y} = f(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_k)$ is decorated `com` if and only if the evaluation of the whole expression was common as defined in §7.4.

Flavors should define other decoration values, but `com` is the only one that is required to have the same meaning in all flavors.

[*Examples. Reasons why an individual evaluation of φ with common inputs $\boldsymbol{x} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k)$ may* not *return* `com` *include the following.*

Outside domain: *The implementation finds φ is not defined and continuous everywhere on $\boldsymbol{x}$. Examples:* $\sqrt{[-4, 4]}$, $\mathrm{sign}([0, 2])$.

Overflow: *The Level 1 result is too large to be represented. Example: Consider an interval type $\mathbb{T}$ whose intervals are represented by their lower and upper bounds in some floating point format, let* `REALMAX` *be the largest finite number in that format, and $\boldsymbol{x}$ be the common $\mathbb{T}$-interval* $[0, \mathtt{REALMAX}]$. *Then $\boldsymbol{x} + \boldsymbol{x}$ cannot be enclosed in a common $\mathbb{T}$-interval.*

Cost: *It is too expensive to determine whether the result can be represented. A possible example is* $\tan([a, b])$ *where* $[a, b]$ *is of a high-precision interval type, and one of its endpoints happens to be very close to a singularity of* $\tan(x)$.

]

CHAPTER 2

Set-Based Intervals

This Chapter contains the standard for the set-based interval flavor.

9. Level 1 description

In this clause, subclauses §9.1 to §9.5 describe the theory of mathematical intervals and interval functions that underlies this flavor. The relation between expressions and the point or interval functions that they define is specified, since it is central to the Fundamental Theorem of Interval Arithmetic. Subclauses §9.6, 9.7 list the required and recommended *arithmetic operations* (also called elementary functions) with their mathematical specifications. Clause 10 describes, at a mathematical level, the system of *decorations* that is used among other things for exception handling in this flavor of the standard.

9.1. Non-interval Level 1 entities. In addition to intervals, this flavor deals with entities of the following kinds. They may be used as inputs or outputs of operations.

– The set $\overline{\mathbb{R}} = \mathbb{R} \cup \{-\infty, +\infty\}$ of **extended reals**. Following the terminology of 754 (e.g., 754§2.1.25), any member of $\overline{\mathbb{R}}$ is called a number: it is a **finite number** if it belongs to $\mathbb{R}$, else an **infinite number**.

  An interval's members are finite numbers, but its bounds can be infinite. Finite or infinite numbers can be inputs to interval constructors, as well as outputs from operations, e.g., the interval width operation.

– The set of **(text) strings**, namely finite sequences of **characters** chosen from some alphabet. Since Level 1 is primarily for human communication, there are no Level 1 restrictions on the alphabet used. Strings may be inputs to interval constructors, as well as inputs or outputs of read/write operations.

9.2. Intervals. The set of mathematical intervals supported by this flavor is denoted $\overline{\mathbb{IR}}$. It consists of exactly those subsets $\boldsymbol{x}$ of the real line $\mathbb{R}$ that are closed and connected in the topological sense. Thus it comprises the empty set (denoted $\emptyset$ or Empty) together with all the nonempty intervals, denoted $[\underline{x}, \overline{x}]$, defined by

$$[\underline{x}, \overline{x}] = \{\, x \in \mathbb{R} \mid \underline{x} \le x \le \overline{x} \,\}, \tag{7}$$

where $\underline{x}$ and $\overline{x}$, the **bounds** of the interval, are extended-real numbers satisfying $\underline{x} \le \overline{x}$, $\underline{x} < +\infty$ and $\overline{x} > -\infty$.

[*Notes.*

– *The above definition implies $-\infty$ and $+\infty$ can be bounds of an interval, but are never members of it.*

  In particular, $[-\infty, +\infty]$ is the set of all real *numbers satisfying $-\infty \le x \le +\infty$, which is the whole real line $\mathbb{R}$—not the whole extended real line $\overline{\mathbb{R}}$.*

– *Mathematical literature generally uses a round bracket, or reversed square bracket, to show that an endpoint is excluded from an interval, e.g. $(a, b]$ and $]a, b]$ denote $\{\, x \mid a < x \le b \,\}$. Where it is convenient to change to this notation, this is pointed out, e.g., in the tables of function domains and ranges in §9.6, 9.7.*

– *The set of intervals $\overline{\mathbb{IR}}$ could be described more concisely as comprising all sets $\{\, x \in \mathbb{R} \mid \underline{x} \le x \le \overline{x} \,\}$ for* arbitrary *extended-real $\underline{x}$, $\overline{x}$. However, this obtains* Empty *in many ways, as $[\underline{x}, \overline{x}]$ for any bounds satisfying $\underline{x} > \overline{x}$, and also as $[-\infty, -\infty]$ or $[+\infty, +\infty]$. The description (7) was preferred as it makes a one-to-one mapping between valid pairs $\underline{x}, \overline{x}$ of endpoints and the nonempty intervals they specify.*

]

A **box** or **interval vector** is an n-tuple $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$ whose components $\boldsymbol{x}_i$ are intervals, that is a member of $\overline{\mathbb{IR}}^n$. Usually $\boldsymbol{x}$ is identified with the cartesian product $\boldsymbol{x}_1 \times \ldots \times \boldsymbol{x}_n$ of its components, a subset of $\mathbb{R}^n$. In particular $x \in \boldsymbol{x}$, for $x \in \mathbb{R}^n$, means by definition $x_i \in \boldsymbol{x}_i$ for all $i = 1, \ldots, n$; and $\boldsymbol{x}$ is empty if (and only if) any of its components $\boldsymbol{x}_i$ is empty.

9.3. Hull. The (interval) **hull** of an arbitrary subset $\boldsymbol{s}$ of $\mathbb{R}^n$, written hull($\boldsymbol{s}$), is the tightest member of $\overline{\mathbb{IR}}^n$ that contains $\boldsymbol{s}$. (The **tightest** set with a given property is the intersection of all sets having that property, provided the intersection itself has this property.)

9.4. Functions.

9.4.1. *Function terminology.* In this flavor, operations are written as named functions; in a specific implementation they might be represented by operators (e.g., using an infix notation), or by families of type-specific functions, or by operators or functions whose names might differ from those used here.

The terms operation, function and mapping are broadly synonymous. The following summarizes usage, with references in parentheses to precise definitions of terms.

– A *point function* (§9.4.2) is a mathematical real function of real variables. Otherwise, *function* is usually used with its general mathematical meaning.
– A (point) *arithmetic operation* (§9.4.2) is a mathematical real function for which an implementation provides versions in the implementation's *library* (§9.4.2).
– A *version* of a point function f means a function derived from f; typically a bare or decorated interval extension (§9.4.3) of f.
– An *interval arithmetic operation* is an interval extension of a point arithmetic operation (§9.4.3).
– An *interval non-arithmetic operation* is an interval-to-interval library function that is not an interval arithmetic operation (§9.4.3).
– A *constructor* is a function that creates an interval from non-interval data (§9.6.8).

9.4.2. *Point functions.* A **point function** is a (possibly partial) multivariate real function: that is, a mapping f from a subset D of $\mathbb{R}^n$ to $\mathbb{R}^m$ for some integers $n \geq 0, m > 0$. It is a *scalar* function if $m = 1$, otherwise a *vector* function. When not otherwise specified, scalar is assumed. The set D where f is defined is its **domain**, also written $\operatorname{Dom} f$. To specify n, call f an n-variable point function, or denote values of f as

$$f(x_1, \ldots, x_n). \tag{8}$$

The **range** of f over an arbitrary subset $\boldsymbol{s}$ of $\mathbb{R}^n$ is the set $\operatorname{Rge}(f \mid \boldsymbol{s})$ defined by

$$\operatorname{Rge}(f \mid \boldsymbol{s}) = \{\, f(x) \mid x \in \boldsymbol{s} \text{ and } x \in \operatorname{Dom} f \,\}. \tag{9}$$

Thus mathematically, when evaluating a function over a set, points outside the domain are ignored—e.g., $\operatorname{Rge}(\mathrm{sqrt} \mid [-1, 1]) = [0, 1]$.

Equivalently, for the case where f takes separate arguments $\boldsymbol{s}_1, \ldots, \boldsymbol{s}_n$, each being a subset of $\mathbb{R}$, the range is written as $\operatorname{Rge}(f \mid \boldsymbol{s}_1, \ldots, \boldsymbol{s}_n)$. This is an alternative notation when $\boldsymbol{s}$ is the cartesian product of the $\boldsymbol{s}_i$.

A (point) **arithmetic operation** is a function for which an implementation provides versions in a collection of user-available operations called its **library**. This includes functions normally written in operator form (e.g., $+$, $\times$) and those normally written in function form (e.g., exp, arctan). It is not specified how an implementation provides library facilities.

9.4.3. *Interval-valued functions.* A box is an interval vector $\boldsymbol{x} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n) \in \overline{\mathbb{IR}}^n$. It is usually identified with the cartesian product $\boldsymbol{x}_1 \times \ldots \times \boldsymbol{x}_n \subseteq \mathbb{R}^n$; however, the correspondence is one-to-one only when all the $\boldsymbol{x}_j$ are nonempty.

Given an n-variable scalar point function f, an **interval extension** of f is a (total) mapping $\boldsymbol{f}$ from n-dimensional boxes to intervals, that is $\boldsymbol{f} : \overline{\mathbb{IR}}^n \to \overline{\mathbb{IR}}$, such that $f(x) \in \boldsymbol{f}(\boldsymbol{x})$ whenever $x \in \boldsymbol{x}$ and $f(x)$ is defined, equivalently

$$\boldsymbol{f}(\boldsymbol{x}) \supseteq \operatorname{Rge}(f \mid \boldsymbol{x})$$

for any box $\boldsymbol{x} \in \overline{\mathbb{IR}}^n$, regarded as a subset of $\mathbb{R}^n$. The **natural interval extension** of f is the mapping $\boldsymbol{f}$ defined by

$$\boldsymbol{f}(\boldsymbol{x}) = \operatorname{hull}(\operatorname{Rge}(f \mid \boldsymbol{x})).$$

Equivalently, using multiple-argument notation for f, an interval extension satisfies

$$\boldsymbol{f}(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n) \supseteq \operatorname{Rge}(f \mid \boldsymbol{x}_1, \ldots, \boldsymbol{x}_n),$$

and the natural interval extension is defined by

$$\boldsymbol{f}(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n) = \text{hull}(\text{Rge}(f \mid \boldsymbol{x}_1, \ldots, \boldsymbol{x}_n))$$

for any intervals $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$.

In some contexts it is useful for $\boldsymbol{x}$ to be a general subset of $\mathbb{R}^n$, or the $\boldsymbol{x}_i$ to be general subsets of $\mathbb{R}$; the definition is unchanged.

The natural extension is automatically defined for all interval or set arguments. The decoration system, Clause 10, gives a way of diagnosing when the underlying point function has been evaluated outside its domain.

When f is a binary operator $\bullet$ written in infix notation, this gives the usual definition of its natural interval extension as

$$\boldsymbol{x} \bullet \boldsymbol{y} = \text{hull}(\{\, x \bullet y \mid x \in \boldsymbol{x},\, y \in \boldsymbol{y}, \text{ and } x \bullet y \text{ is defined} \,\}).$$

[*Example. With these definitions, the relevant natural interval extensions satisfy* $\sqrt{[-1,4]} = [0,2]$ *and* $\sqrt{[-2,-1]} = \emptyset$*; also* $\boldsymbol{x} \times [0,0] = [0,0]$ *for any nonempty* $\boldsymbol{x}$*, and* $\boldsymbol{x}/[0,0] = \emptyset$*, for any* $\boldsymbol{x}$.]

When f is a vector point function, a vector interval function with the same number of inputs and outputs as f is called an interval extension of f if each of its components is an interval extension of the corresponding component of f.

An interval-valued function in the library is called an **interval arithmetic operation** if it is an interval extension of a point arithmetic operation, and an **interval non-arithmetic operation** otherwise. Examples of the latter are interval intersection and union, $(\boldsymbol{x}, \boldsymbol{y}) \mapsto \boldsymbol{x} \cap \boldsymbol{y}$ and $(\boldsymbol{x}, \boldsymbol{y}) \mapsto \text{hull}(\boldsymbol{x} \cup \boldsymbol{y})$.

9.4.4. *Constants.* A real scalar function with no arguments—a mapping $\mathbb{R}^n \to \mathbb{R}^m$ with $n = 0$ and $m = 1$—is a **real constant**. Languages may distinguish between a literal constant (e.g., the decimal value defined by the string `1.23e4`) and a named constant (e.g., π) but the difference is not relevant on Level 1 (and easily handled by outward rounding on Level 2).

From the definition, an interval extension of a real constant is any zero-argument interval function that returns an interval containing c. The *natural extension* returns the interval $[c, c]$.

9.5. Expressions. This flavor gives the term "expression" the general meaning described in Clause 6.

9.6. Required operations.

For the functions listed in this subclause, an implementation shall provide interval versions appropriate to its supported interval types. For constants and the forward and reverse arithmetic operations in §9.6.1, 9.6.3, 9.6.4, 9.6.5, each such version shall be an interval extension (§9.4.3) of the corresponding point function—for a constant, that means any constant interval enclosing the point value. The required rounding behavior of these, and of the numeric functions of intervals in §9.6.9, is detailed in §11.9, 11.11.

The names of operations, as well as symbols used for operations (e.g., for the comparisons in §9.6.10), may not correspond to those that any particular language would use.

9.6.1. *Interval literals.*

An implementation shall provide denotations of exact interval values by text strings. These are called **interval literals**. Level 1, which is mainly for human communication, makes no requirements on the form of literals. This document uses the Level 2 syntax, specified in §11.11.1. [*Example. This includes the inf-sup form* `[1.234e5,Inf]`*; the mid-rad form* `<3.1416+-0.00001>`*; or the named interval constant* `Entire`.]

An invalid denotation has no value at Level 1.

9.6.2. *Interval constants.*

The constant functions `bareempty`() and `bareentire`() return Empty and Entire respectively.

9.6.3. *Forward-mode elementary functions.*

Table 1 on page 21 lists required arithmetic operations. The term *operation* includes functions normally written in function notation $f(x, y, \ldots)$, as well as those normally written in unary or binary operator notation, $\bullet x$ or $x \bullet y$.
[*Note. The list includes all general-computational operations in 754§5.4 except* **convertFromInt**, *and some recommended functions in 754§9.2.*]

Proving the correctness of interval computations relies on the Fundamental Theorem of Interval Arithmetic, which in turn relies on the relation between a point-function and its interval extensions. Thus the domain of each point-function and its value at each point of the domain are specified to aid a rigorous implementation. This is mostly straightforward but needs care for functions with discontinuities, such as pow() and atan2().

Notes to Table 1

a. In describing the domain, notation such as $\{y = 0\}$ is short for $\{\, (x, y) \in \mathbb{R}^2 \mid y = 0 \,\}$, etc.
b. Regarded as a family of functions parameterized by the integer argument p.
c. Defined as $e^{y \ln x}$ for real $x > 0$ and all real y, and 0 for $x = 0$ and $y > 0$, else there is no value at Level 1.
d. $b = e, 2$ or 10, respectively.
e. The ranges shown are the mathematical range of the point function. To ensure containment, an interval result may include values just outside the mathematical range.
f. atan2(y, x) is the principal value of the argument (polar angle) of (x, y) in the plane.
g. To avoid confusion with notation for open intervals, in this table coordinates in $\mathbb{R}^2$ are delimited by angle brackets $\langle\ \rangle$.
h. `sign`(x) is -1 if $x < 0$; 0 if $x = 0$; and 1 if $x > 0$.
i. `ceil`(x) is the smallest integer $\geq x$. `floor`(x) is the largest integer $\leq x$. `roundTiesToEven`(x), `roundTiesToAway`(x) are the nearest integer to x, with ties rounded to the even integer or away from zero respectively. `trunc`(x) is the nearest integer to x in the direction of zero. (As defined in the C standard §7.12.9.)
j. Smallest, or largest, of its real arguments. A family of functions parameterized by the arity k.

9.6.4. *Interval case expressions and case function.*

Functions are often defined by conditionals: $f(x)$ equals $g(x)$ if some condition on x holds, and $h(x)$ otherwise. To handle interval extensions of such functions in a way that automatically conforms to the Fundamental Theorem of Interval Arithmetic, the ternary function `case`(c, g, h) is provided. To simplify defining its interval extension, the argument c specifying the condition is real (instead of boolean), and the condition means $c < 0$ by definition. That is,

$$\texttt{case}(c, g, h) = \begin{cases} \text{if } c < 0 & \text{then} \quad g, \\ & \text{else} \quad h. \end{cases}$$

TABLE 1. Required forward elementary functions.
Normal mathematical notation is used to include or exclude an interval endpoint, e.g., $(-\pi, \pi]$ denotes $\{ x \in \mathbb{R} \mid -\pi < x \leq \pi \}$.

Name	Definition	Point function domain	Point function range	Note
$\texttt{neg}(x)$	$-x$	$\mathbb{R}$	$\mathbb{R}$	
$\texttt{add}(x, y)$	$x + y$	$\mathbb{R}^2$	$\mathbb{R}$	
$\texttt{sub}(x, y)$	$x - y$	$\mathbb{R}^2$	$\mathbb{R}$	
$\texttt{mul}(x, y)$	xy	$\mathbb{R}^2$	$\mathbb{R}$	
$\texttt{div}(x, y)$	x/y	$\mathbb{R}^2 \setminus \{y = 0\}$	$\mathbb{R}$	a
$\texttt{recip}(x)$	$1/x$	$\mathbb{R} \setminus \{0\}$	$\mathbb{R} \setminus \{0\}$	
$\texttt{sqrt}(x)$	$\sqrt{x}$	$[0, \infty)$	$[0, \infty)$	
$\texttt{fma}(x, y, z)$	$(x \times y) + z$	$\mathbb{R}^3$	$\mathbb{R}$	
$\texttt{case}(c, g, h)$		See §9.6.4.		
$\texttt{sqr}(x)$	x^2	$\mathbb{R}$	$[0, \infty)$	
$\texttt{pown}(x, p)$	$x^p,\ p \in \mathbb{Z}$	$\begin{cases} \mathbb{R} \text{ if } p \geq 0 \\ \mathbb{R}\backslash\{0\} \text{ if } p < 0 \end{cases}$	$\begin{cases} \mathbb{R} \text{ if } p > 0 \text{ odd} \\ [0, \infty) \text{ if } p > 0 \text{ even} \\ \{1\} \text{ if } p = 0 \\ \mathbb{R}\backslash\{0\} \text{ if } p < 0 \text{ odd} \\ (0, \infty) \text{ if } p < 0 \text{ even} \end{cases}$	b
$\texttt{pow}(x, y)$	x^y	$\{x{>}0\} \cup \{x{=}0, y{>}0\}$	$[0, \infty)$	a, c
$\texttt{exp,exp2,exp10}(x)$	b^x	$\mathbb{R}$	$(0, \infty)$	d
$\texttt{log,log2,log10}(x)$	$\log_b x$	$(0, \infty)$	$\mathbb{R}$	d
$\texttt{sin}(x)$		$\mathbb{R}$	$[-1, 1]$	
$\texttt{cos}(x)$		$\mathbb{R}$	$[-1, 1]$	
$\texttt{tan}(x)$		$\mathbb{R}\backslash\{(k + \frac{1}{2})\pi \mid k \in \mathbb{Z}\}$	$\mathbb{R}$	
$\texttt{asin}(x)$		$[-1, 1]$	$[-\pi/2, \pi/2]$	e
$\texttt{acos}(x)$		$[-1, 1]$	$[0, \pi]$	e
$\texttt{atan}(x)$		$\mathbb{R}$	$(-\pi/2, \pi/2)$	e
$\texttt{atan2}(y, x)$		$\mathbb{R}^2 \setminus \{\langle 0, 0 \rangle\}$	$(-\pi, \pi]$	e, f, g
$\texttt{sinh}(x)$		$\mathbb{R}$	$\mathbb{R}$	
$\texttt{cosh}(x)$		$\mathbb{R}$	$[1, \infty)$	
$\texttt{tanh}(x)$		$\mathbb{R}$	$(-1, 1)$	
$\texttt{asinh}(x)$		$\mathbb{R}$	$\mathbb{R}$	
$\texttt{acosh}(x)$		$[1, \infty)$	$[0, \infty)$	
$\texttt{atanh}(x)$		$(-1, 1)$	$\mathbb{R}$	
$\texttt{sign}(x)$		$\mathbb{R}$	$\{-1, 0, 1\}$	h
$\texttt{ceil}(x)$		$\mathbb{R}$	$\mathbb{Z}$	i
$\texttt{floor}(x)$		$\mathbb{R}$	$\mathbb{Z}$	i
$\texttt{trunc}(x)$		$\mathbb{R}$	$\mathbb{Z}$	i
$\texttt{roundTiesToEven}(x)$		$\mathbb{R}$	$\mathbb{Z}$	i
$\texttt{roundTiesToAway}(x)$		$\mathbb{R}$	$\mathbb{Z}$	i
$\texttt{abs}(x)$	$\lvert x \rvert$	$\mathbb{R}$	$[0, \infty)$	
$\texttt{min}(x_1, \ldots, x_k)$		$\mathbb{R}^k$ for $k = 2, 3, \ldots$	$\mathbb{R}$	j
$\texttt{max}(x_1, \ldots, x_k)$		$\mathbb{R}^k$ for $k = 2, 3, \ldots$	$\mathbb{R}$	j

An implementation shall provide the following interval extension (see the Notes):

$$\texttt{case}(\boldsymbol{c}, \boldsymbol{g}, \boldsymbol{h}) = \begin{cases} \text{if } \boldsymbol{c} \text{ is empty} & \text{then} & \emptyset \\ \text{elseif } \boldsymbol{c} \text{ is a subset of the half- line } x < 0 & \text{then} & \boldsymbol{g} \\ \text{elseif } \boldsymbol{c} \text{ is a subset of the half- line } x \geq 0 & \text{then} & \boldsymbol{h} \\ \text{else} & & \texttt{convexHull}(\boldsymbol{g}, \boldsymbol{h}). \end{cases} \tag{10}$$

for any intervals $\boldsymbol{c}, \boldsymbol{g}, \boldsymbol{h}$.

The function f above may be encoded as $f(x) = \mathtt{case}\big(c(x), g(x), h(x)\big)$. Then, if $\boldsymbol{c}$, $\boldsymbol{g}$, $\boldsymbol{h}$ are interval functions that are interval extensions of point functions c, g and h, the function

$$\boldsymbol{f}(\boldsymbol{x}) = \mathtt{case}\big(\boldsymbol{c}(\boldsymbol{x}), \boldsymbol{g}(\boldsymbol{x}), \boldsymbol{h}(\boldsymbol{x})\big) \tag{11}$$

is automatically an interval extension of f.
[*Notes.*

1. *Equation (10) does not define the natural interval extension, which returns Empty if* any *of its input arguments is empty. Its advantage is that for a function defined by a conditional expression, such as (11), it allows "short-circuiting". That is, one can suppress evaluation of $\boldsymbol{h}(\boldsymbol{x})$ if $\overline{c} < 0$, and of $\boldsymbol{g}(\boldsymbol{x})$ if $\underline{c} \geq 0$. This is not so for the natural extension.*
2. *This method is less awkward than using interval comparisons as a mechanism for handling such functions. However, the resulting interval function is usually not the tightest extension of the corresponding point function. E.g., the (point) absolute value $|x|$ may be defined by*

   $$|x| = \mathtt{case}(x, -x, x).$$

   Then it is easy to see that formula (11), applied to a nonempty $\boldsymbol{x} = [\underline{x}, \overline{x}]$, gives the exact range $\{\, |x| \mid x \in \boldsymbol{x} \,\}$ when $\overline{x} < 0$ or $0 \leq \underline{x}$, but the poor enclosure $(-\boldsymbol{x}) \cup \boldsymbol{x}$ when $\underline{x} < 0 \leq \overline{x}$.
3. $\mathtt{case}(c, g, h)$ *is equivalent to the C expression $(c < 0\,?\,g : h)$.*
4. *Compound conditions may be expressed using the* `max` *and* `min` *operations: e.g., a real function $f(x, y)$ that equals* $\sin(xy)$ *in the positive quadrant of the plane, and zero elsewhere, may be written*

   $$f(x, y) = \mathtt{case}(\min(x, y), 0, \sin(xy)),$$

   since $\min(x, y) < 0$ *is equivalent to* $(x < 0$ *or* $y < 0)$.

   ]

9.6.5. *Reverse-mode elementary functions.*
Constraint-satisfaction algorithms use the functions in this subclause for iteratively tightening an enclosure of a solution to a system of equations.
Given a unary arithmetic operation φ, a **reverse interval extension** of φ is a binary interval function φRev such that

$$\varphi\mathrm{Rev}(\boldsymbol{c}, \boldsymbol{x}) \supseteq \{\, x \in \boldsymbol{x} \mid \varphi(x) \text{ is defined and in } \boldsymbol{c} \,\}, \tag{12}$$

for any intervals $\boldsymbol{c}, \boldsymbol{x}$.
Similarly, a binary arithmetic operation $\bullet$ has two forms of reverse interval extension, which are ternary interval functions $\bullet\mathrm{Rev}_1$ and $\bullet\mathrm{Rev}_2$ such that

$$\bullet\mathrm{Rev}_1(\boldsymbol{b}, \boldsymbol{c}, \boldsymbol{x}) \supseteq \{\, x \in \boldsymbol{x} \mid b \in \boldsymbol{b} \text{ exists such that } x \bullet b \text{ is defined and in } \boldsymbol{c} \,\}, \tag{13}$$

$$\bullet\mathrm{Rev}_2(\boldsymbol{a}, \boldsymbol{c}, \boldsymbol{x}) \supseteq \{\, x \in \boldsymbol{x} \mid a \in \boldsymbol{a} \text{ exists such that } a \bullet x \text{ is defined and in } \boldsymbol{c} \,\}. \tag{14}$$

If $\bullet$ is commutative then $\bullet\mathrm{Rev}_1$ and $\bullet\mathrm{Rev}_2$ agree and may be implemented simply as $\bullet$Rev.
In each of (12, 13, 14), the unique **natural reverse interval extension** is the one whose value is the interval hull of the right-hand side. Clearly, any reverse interval extension encloses this hull.
The last argument $\boldsymbol{x}$ in each of (12, 13, 14) is optional, with default $\boldsymbol{x} = \mathbb{R}$ if absent.
[*Note. The argument $\boldsymbol{x}$ can be thought of as giving prior knowledge about the range of values taken by a point-variable x, which is then sharpened by applying the reverse function: see the example below.*]
Reverse operations shall be provided as in Table 2. Note `pownRev`(x, p) is regarded as a family of unary functions parametrized by p.
[*Example.*

– *Consider the function $sqr(x) = x^2$. Evaluating sqr*Rev$([1, 4])$ *answers the question: given that $1 \leq x^2 \leq 4$, what interval can we restrict x to? Using the natural reverse extension, we have*

$$sqr\mathrm{Rev}([1, 4]) = \mathsf{hull}\{\, x \in \mathbb{R} \mid x^2 \in [1, 4] \,\} = \mathsf{hull}([-2, -1] \cup [1, 2]) = [-2, 2].$$

– *If we can add the prior knowledge that $x \in \boldsymbol{x} = [0, 1.2]$, then using the optional second argument gives the tighter enclosure*

$$sqr\mathrm{Rev}([1, 4], [0, 1.2]) = \mathsf{hull}\{\, x \in [0, 1.2] \mid x^2 \in [1, 4] \,\} = \mathsf{hull}\big([0, 1.2] \cap ([-2, -1] \cup [1, 2])\big) = [1, 1.2].$$

TABLE 2. Required reverse elementary functions.

From unary functions	From binary functions
`sqrRev`$(\boldsymbol{c}, \boldsymbol{x})$	`mulRev`$(\boldsymbol{b}, \boldsymbol{c}, \boldsymbol{x})$
`recipRev`$(\boldsymbol{c}, \boldsymbol{x})$	`divRev1`$(\boldsymbol{b}, \boldsymbol{c}, \boldsymbol{x})$
`absRev`$(\boldsymbol{c}, \boldsymbol{x})$	`divRev2`$(\boldsymbol{a}, \boldsymbol{c}, \boldsymbol{x})$
`pownRev`$(\boldsymbol{c}, \boldsymbol{x}, p)$	`powRev1`$(\boldsymbol{b}, \boldsymbol{c}, \boldsymbol{x})$
`sinRev`$(\boldsymbol{c}, \boldsymbol{x})$	`powRev2`$(\boldsymbol{a}, \boldsymbol{c}, \boldsymbol{x})$
`cosRev`$(\boldsymbol{c}, \boldsymbol{x})$	`atan2Rev1`$(\boldsymbol{b}, \boldsymbol{c}, \boldsymbol{x})$
`tanRev`$(\boldsymbol{c}, \boldsymbol{x})$	`atan2Rev2`$(\boldsymbol{a}, \boldsymbol{c}, \boldsymbol{x})$
`coshRev`$(\boldsymbol{c}, \boldsymbol{x})$	

– *One might think it suffices to apply the operation without the optional argument and intersect the result with $\boldsymbol{x}$. This is less effective because "hull" and "intersect" do not commute. E.,g., in the above, this method evaluates*

$$sqr\mathrm{Rev}([1,4]) \cap \boldsymbol{x} = [-2,2] \cap [0,1.2] = [0,1.2],$$

so no tightening of the enclosure $\boldsymbol{x}$ is obtained.

]

9.6.6. *Cancellative addition and subtraction.*

⚠ I have made this only apply to bounded intervals, since it really seems hard to frame a specification for unbounded ones that translates unambiguously to the finite precision (Level 2) situation.

Also I have deviated from the Motion 12 spec, by making the operations defined only when `width`$(\boldsymbol{x}) \geq$ `width`$(\boldsymbol{y})$. IMO it is actively harmful in applications to make it always defined. This is for the same reasons that it is harmful to replace $\sqrt{x}$ by the everywhere defined $\sqrt{|x|}$: an application MUST test for definedness, and making it always defined leads to un-noticed wrong results from code that forgets to test. With Kaucher/modal intervals a different choice may be appropriate.

Cancellative subtraction solves the problem: Recover interval $\boldsymbol{z}$ from intervals $\boldsymbol{x}$ and $\boldsymbol{y}$, given that one knows $\boldsymbol{x}$ was obtained as the sum $\boldsymbol{y} + \boldsymbol{z}$.

[*Example. In some applications one has a list of intervals $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n$, and needs to form each interval $\boldsymbol{s}_k$ which is the sum of all the $\boldsymbol{a}_i$ except $\boldsymbol{a}_k$, that is $\boldsymbol{s}_k = \sum_{i=1,\, i\neq k}^{n} \boldsymbol{a}_i$, for $k = 1, \ldots, n$. Evaluating all these sums independently costs $O(n^2)$ work. However, if one forms the sum $\boldsymbol{s}$ of* all *the $\boldsymbol{a}_i$, one can obtain each $\boldsymbol{s}_k$ from $\boldsymbol{s}$ and $\boldsymbol{a}_k$ by cancellative subtraction. This method only costs $O(n)$ work.*

This example illustrates that in finite precision, computing $\boldsymbol{x}$ (as a sum of terms) typically incurs at least one roundoff error, and may incur many. Thus the model underlying these cancellative operations is that $\boldsymbol{x}$ is an enclosure of an unknown true sum $\boldsymbol{x}_0$, whereas y is "exact". The computed $\boldsymbol{z}$ is thus an enclosure of an unknown true $\boldsymbol{z}_0$ such that $\boldsymbol{y} + \boldsymbol{z}_0 = \boldsymbol{x}_0$.]

There is an operation `cancelPlus`$(\boldsymbol{x}, \boldsymbol{y})$, equivalent to `cancelMinus`$(\boldsymbol{x}, -\boldsymbol{y})$ and therefore not specified separately.

There is an operation `cancelMinus`$(\boldsymbol{x}, \boldsymbol{y})$ that returns for any two bounded intervals $\boldsymbol{x}$ and $\boldsymbol{y}$ the tightest interval $\boldsymbol{z}$ such that

$$\boldsymbol{y} + \boldsymbol{z} \supseteq \boldsymbol{x} \tag{15}$$

if such a $\boldsymbol{z}$ exists. Otherwise `cancelMinus`$(\boldsymbol{x}, \boldsymbol{y})$ has no value at Level 1.

This specification leads to the following Level 1 algorithm. If $\boldsymbol{x} = \emptyset$ then $\boldsymbol{z} = \emptyset$. If $\boldsymbol{x} \neq \emptyset$ and $\boldsymbol{y} = \emptyset$ then $\boldsymbol{z}$ has no value. If $\boldsymbol{x} = [\underline{x}, \overline{x}]$ and $\boldsymbol{y} = [\underline{y}, \overline{y}]$ are both nonempty and bounded, define $\underline{z} = \underline{x} - \underline{y}$ and $\overline{z} = \overline{x} - \overline{y}$. Then $\boldsymbol{z}$ is defined to be $[\underline{z}, \overline{z}]$ if $\underline{z} \leq \overline{z}$ (equivalently if `width`$(\boldsymbol{x}) \geq$ `width`$(\boldsymbol{y})$), and has no value otherwise. If either $\boldsymbol{x}$ or $\boldsymbol{y}$ is unbounded, $\boldsymbol{z}$ has no value.

[*Note. Because of the cancellative nature of these operations, care is needed in finite precision to determine whether the result is defined or not. More details are given at Level 3 in §14.5.*]

9.6.7. *Non-arithmetic (set) operations.*

The following operations shall be provided, the arguments and result being intervals.

Name	Definition
`intersection`$(\boldsymbol{x},\boldsymbol{y})$	$\boldsymbol{x}\cap\boldsymbol{y}$
`convexHull`$(\boldsymbol{x},\boldsymbol{y})$	interval hull of $\boldsymbol{x}\cup\boldsymbol{y}$

9.6.8. *Constructors.*

An interval constructor by definition is an operation that creates a bare or decorated interval from non-interval data. The following bare interval constructors shall be provided.

The operation `nums2interval`(l,u), where l and u are extended-real values, returns the set $\{\, x\in\mathbb{R} \mid l\le x\le u\,\}$. If (see §9.2) the conditions $l\le u$, $l<+\infty$ and $u>-\infty$ hold, this set is the nonempty interval $[l,u]$ and the operation is said to *succeed*. Otherwise the operation is said to *fail*, and returns no value.

The operation `text2interval`(t) succeeds and returns the interval denoted by the text string t, if t denotes an interval. Otherwise, it fails and returns no value.

[*Note. Since Level 1 is mainly for human-human communication, any understandable* t *is acceptable, e.g.* "`[3.1,4.2]`" *or* "`[`2π`,`∞`]`"*. Rules for the strings* t *accepted at an implementation level are given in the Level 2 Subclause 13 on I/O and may optionally be followed.*]

9.6.9. *Numeric functions of intervals.*

The operations in Table 3 shall be provided, the argument being an interval and the result a number, which for some of the operations may be infinite.

[*Note. Implementations should provide an operation that returns* `mid`$(\boldsymbol{x})$ *and* `rad`$(\boldsymbol{x})$ *simultaneously.*]

TABLE 3. Required numeric functions of an interval $\boldsymbol{x}=[\underline{x},\overline{x}]$.
Note `inf` can return $-\infty$; each of `sup`, `wid`, `rad` and `mag` can return $+\infty$.

Name	Definition
`inf`$(\boldsymbol{x})$	$\begin{cases}\text{lower bound of } \boldsymbol{x} \text{ if } \boldsymbol{x} \text{ is nonempty}\\ \infty \text{ if } \boldsymbol{x} \text{ is empty}\end{cases}$
`sup`$(\boldsymbol{x})$	$\begin{cases}\text{upper bound of } \boldsymbol{x} \text{ if } \boldsymbol{x} \text{ is nonempty}\\ -\infty \text{ if } \boldsymbol{x} \text{ is empty}\end{cases}$
`mid`$(\boldsymbol{x})$	$\begin{cases}\text{midpoint } (\underline{x}+\overline{x})/2 \text{ if } \boldsymbol{x} \text{ is nonempty bounded}\\ \text{no value if } \boldsymbol{x} \text{ is empty or unbounded}\end{cases}$
`wid`$(\boldsymbol{x})$	$\begin{cases}\text{width } \overline{x}-\underline{x} \text{ if } \boldsymbol{x} \text{ is nonempty}\\ \text{no value if } \boldsymbol{x} \text{ is empty}\end{cases}$
`rad`$(\boldsymbol{x})$	$\begin{cases}\text{radius } (\overline{x}-\underline{x})/2 \text{ if } \boldsymbol{x} \text{ is nonempty}\\ \text{no value if } \boldsymbol{x} \text{ is empty}\end{cases}$
`mag`$(\boldsymbol{x})$	$\begin{cases}\text{magnitude } \sup\{\,\lvert x\rvert \mid x\in\boldsymbol{x}\,\} \text{ if } \boldsymbol{x} \text{ is nonempty}\\ \text{no value if } \boldsymbol{x} \text{ is empty}\end{cases}$
`mig`$(\boldsymbol{x})$	$\begin{cases}\text{mignitude } \inf\{\,\lvert x\rvert \mid x\in\boldsymbol{x}\,\} \text{ if } \boldsymbol{x} \text{ is nonempty}\\ \text{no value if } \boldsymbol{x} \text{ is empty}\end{cases}$

⚠ I have gone for simplicity. Also I have followed my own view that at Level 1 it is more consistent with math conventions for a function to simply have "no value" outside its domain, than for it to return a special value. At Level 2 this translates into returning a value such as NaN.

9.6.10. *Boolean functions of intervals.*

The following operations shall be provided, which return a boolean (1 = true, 0 = false) result.

There is a function `isEmpty`$(\boldsymbol{x})$, which returns 1 if $\boldsymbol{x}$ is the empty set, 0 otherwise. There is a function `isEntire`$(\boldsymbol{x})$, which returns 1 if $\boldsymbol{x}$ is the whole line, 0 otherwise.

There are eight comparison relations, which take two interval inputs and return a boolean result. These are defined in Table 4, in which column three gives the set-theoretic definition, and column four gives an equivalent specification when both intervals are nonempty.

The following table shows what the definitions imply when at least one interval is empty.

Table 4. Comparisons for intervals $\boldsymbol{a}$ and $\boldsymbol{b}$. Notation $\forall_a$ means "for all a in $\boldsymbol{a}$", and so on. In column 4, $\boldsymbol{a}=[\underline{a},\overline{a}]$ and $\boldsymbol{b}=[\underline{b},\overline{b}]$, where $\underline{a},\underline{b}$ may be $-\infty$ and $\overline{a},\overline{b}$ may be $+\infty$; and $<'$ is the same as $<$ except that $-\infty <' -\infty$ and $+\infty <' +\infty$ are true.

Name	Symbol	Definition	For $\boldsymbol{a},\boldsymbol{b} \neq \emptyset$	Description
`equal`$(\boldsymbol{a},\boldsymbol{b})$	$\boldsymbol{a} = \boldsymbol{b}$	$\forall_a \exists_b\, a = b \wedge \forall_b \exists_a\, b = a$	$\underline{a} = \underline{b} \wedge \overline{a} = \overline{b}$	$\boldsymbol{a}$ equals $\boldsymbol{b}$
`containedIn`$(\boldsymbol{a},\boldsymbol{b})$	$\boldsymbol{a} \subseteq \boldsymbol{b}$	$\forall_a \exists_b\, a = b$	$\underline{b} \leq \underline{a} \wedge \overline{a} \leq \overline{b}$	$\boldsymbol{a}$ is a subset of $\boldsymbol{b}$
`less`$(\boldsymbol{a},\boldsymbol{b})$	$\boldsymbol{a} \leq \boldsymbol{b}$	$\forall_a \exists_b\, a \leq b \wedge \forall_b \exists_a\, a \leq b$	$\underline{a} \leq \underline{b} \wedge \overline{a} \leq \overline{b}$	$\boldsymbol{a}$ is weakly less than $\boldsymbol{b}$
`precedes`$(\boldsymbol{a},\boldsymbol{b})$	$\boldsymbol{a} \preccurlyeq \boldsymbol{b}$	$\forall_a \forall_b\, a \leq b$	$\overline{a} \leq \underline{b}$	$\boldsymbol{a}$ is to left of but may touch $\boldsymbol{b}$
`isInterior`$(\boldsymbol{a},\boldsymbol{b})$	$\boldsymbol{a} \Subset \boldsymbol{b}$	$\forall_a \exists_b\, a < b \wedge \forall_a \exists_b\, b < a$	$\underline{b} <' \underline{a} \wedge \overline{a} <' \overline{b}$	$\boldsymbol{a}$ is interior to $\boldsymbol{b}$
`strictlyLess`$(\boldsymbol{a},\boldsymbol{b})$	$\boldsymbol{a} < \boldsymbol{b}$	$\forall_a \exists_b\, a < b \wedge \forall_b \exists_a\, a < b$	$\underline{a} <' \underline{b} \wedge \overline{a} <' \overline{b}$	$\boldsymbol{a}$ is strictly less than $\boldsymbol{b}$
`strictlyPrecedes`$(\boldsymbol{a},\boldsymbol{b})$	$\boldsymbol{a} \prec \boldsymbol{b}$	$\forall_a \forall_b\, a < b$	$\overline{a} < \underline{b}$	$\boldsymbol{a}$ is strictly to left of $\boldsymbol{b}$
`areDisjoint`$(\boldsymbol{a},\boldsymbol{b})$	$\boldsymbol{a} \mathbin{\cap\!\!\!\!/} \boldsymbol{b}$	$\forall_a \forall_b\, a \neq b$	$\overline{a} < \underline{b} \vee \overline{b} < \underline{a}$	$\boldsymbol{a}$ and $\boldsymbol{b}$ are disjoint

	$\boldsymbol{a} = \emptyset$ $\boldsymbol{b} \neq \emptyset$	$\boldsymbol{a} \neq \emptyset$ $\boldsymbol{b} = \emptyset$	$\boldsymbol{a} = \emptyset$ $\boldsymbol{b} = \emptyset$
$\boldsymbol{a} = \boldsymbol{b}$	0	0	1
$\boldsymbol{a} \subseteq \boldsymbol{b}$	1	0	1
$\boldsymbol{a} \leq \boldsymbol{b}$	0	0	1
$\boldsymbol{a} \preccurlyeq \boldsymbol{b}$	1	1	1
$\boldsymbol{a} \Subset \boldsymbol{b}$	1	0	1
$\boldsymbol{a} < \boldsymbol{b}$	0	0	1
$\boldsymbol{a} \prec \boldsymbol{b}$	1	1	1
$\boldsymbol{a} \mathbin{\cap\!\!\!\!/} \boldsymbol{b}$	1	1	1

[*Note.*
- *Column two gives suggested symbols for use in typeset algorithms.*
- *All these relations, except* $\boldsymbol{a} \mathbin{\cap\!\!\!\!/} \boldsymbol{b}$, *are transitive for* nonempty *intervals.*
- *The first three are reflexive.*
- `isInterior` *uses the topological definition:* $\boldsymbol{b}$ *is a neighbourhood of each point of* $\boldsymbol{a}$. *This implies, for instance, that* `isInterior`(*Entire*,*Entire*) *is true.*
- *In fact all occurrences of* $<$ *in column 4 of Table 4 can be replaced by* $<'$.

]

9.6.11. *Dot product function.*

For point vectors $x = (x_1, \ldots, x_n)$ and $y = (y_1, \ldots, y_n)$ the function `dotProduct`$(x, y) = \sum_{i=1}^{n} x_i y_i$ is defined, but has no requirements at Level 1. It is specified in Subclause 11.11.11 which deals with the Complete Arithmetic datatype and operations.

9.7. Recommended operations (optional).

An implementation should provide interval versions of the functions listed in this subclause. If such an interval version is provided, it shall behave as specified here.

9.7.1. *Forward-mode elementary functions.*

The list of recommended functions is in Table 5. Each interval version shall be an interval extension of the point function.

TABLE 5. Recommended elementary functions.
Normal mathematical notation is used to include or exclude an interval endpoint, e.g., $(-1,1]$ denotes $\{ x \in \mathbb{R} \mid -1 < x \leq 1 \}$.

Name	Definition	Point function domain	Point function range	Note
`rootn`(x,q)	real $\sqrt[q]{x}$, $q \in \mathbb{Z} \setminus \{0\}$	$\begin{cases} \mathbb{R} \text{ if } q > 0 \text{ odd} \\ [0,\infty) \text{ if } q > 0 \text{ even} \\ \mathbb{R}\backslash\{0\} \text{ if } q < 0 \text{ odd} \\ (0,\infty) \text{ if } q < 0 \text{ even} \end{cases}$	same as domain	a
`expm1`(x), `exp2m1`(x), `exp10m1`(x)	$b^x - 1$	$\mathbb{R}$	$(-1,\infty)$	b, c
`logp1`(x), `log2p1`(x), `log10p1`(x)	$\log_b(x+1)$	$(-1,\infty)$	$\mathbb{R}$	b, c
`compoundm1`(x,y)	$(1+x)^y - 1$	$\{x>-1\} \cup \{x=-1, y>0\}$	$[0,\infty)$	c, d
`hypot`(x,y)	$\sqrt{x^2+y^2}$	$\mathbb{R}^2$	$[0,\infty)$	
`rSqrt`(x)	$1/\sqrt{x}$	$(0,\infty)$	$(0,\infty)$	
`sinPi`(x)	$\sin(\pi x)$	$\mathbb{R}$	$[-1,1]$	e
`cosPi`(x)	$\cos(\pi x)$	$\mathbb{R}$	$[-1,1]$	e
`tanPi`(x)	$\tan(\pi x)$	$\mathbb{R}\backslash\{ k+\frac{1}{2} \mid k \in \mathbb{Z} \}$	$\mathbb{R}$	e
`asinPi`(x)	$\arcsin(x)/\pi$	$[-1,1]$	$[-1/2,1/2]$	e
`acosPi`(x)	$\arccos(x)/\pi$	$[-1,1]$	$[0,1]$	e
`atanPi`(x)	$\arctan(x)/\pi$	$\mathbb{R}$	$(-1/2,1/2)$	e
`atan2Pi`(y,x)	$\operatorname{atan2}(y,x)/\pi$	$\mathbb{R}^2 \setminus \{\langle 0,0 \rangle\}$	$(-1,1]$	e, f

Notes to Table 5

a. Regarded as a family of functions parameterized by the integer arguments q, or r and s.
b. $b = e, 2$ or 10, respectively.
c. Mathematically unnecessary, but included to let implementations give better numerical behavior for small values of the arguments.
d. In describing domains, notation such as $\{y = 0\}$ is short for $\{ (x,y) \in \mathbb{R}^2 \mid y = 0 \}$, and so on.
e. These functions avoid a loss of accuracy due to π being irrational, cf. Table 1, note e.
f. To avoid confusion with notation for open intervals, in this table coordinates in $\mathbb{R}^2$ are delimited by angle brackets $\langle\ \rangle$.

9.7.2. *Extended interval comparisons.*

The **interval overlapping operation** `overlap`$(\boldsymbol{a}, \boldsymbol{b})$, also written $\boldsymbol{a} ⦶ \boldsymbol{b}$, arises from the work of J.F. Allen [**1**] on temporal logic. It may be used as an infrastructure for other interval comparisons. If implemented, it should also be available at user level; how this is done is implementation-defined or language-defined.

Allen identified 13 states of a pair $(\boldsymbol{a}, \boldsymbol{b})$ of nonempty intervals, which are ways in which they can be related with respect to the usual order $a < b$ of the reals. Together with three states for when either interval is empty, these define the 16 possible values of `overlap`$(\boldsymbol{a}, \boldsymbol{b})$.

To describe the states for nonempty intervals of positive width, it is useful to think of $\boldsymbol{b} = [\underline{b}, \overline{b}]$ (with $\underline{b} < \overline{b}$) as fixed, while $\boldsymbol{a} = [\underline{a}, \overline{a}]$ (with $\underline{a} < \overline{a}$) starts far to its left and moves to the right. Its endpoints move continuously with strictly positive velocity. Then, depending on the relative sizes of $\boldsymbol{a}$ and $\boldsymbol{b}$, the value of $\boldsymbol{a} ⦶ \boldsymbol{b}$ follows a path from left to right through the graph below, whose nodes represent Allen's 13 states.

$$\begin{tikzcd}
& & & \text{starts} \arrow[r] & \text{containedBy} \arrow[r] & \text{finishes} \arrow[d] \\
{} \arrow[r] & \text{before} \arrow[r] & \text{meets} \arrow[r] & \text{overlaps} \arrow[u] \arrow[d] \arrow[r] & \text{equals} \arrow[r] & \text{overlappedBy} \arrow[r] & \text{metBy} \arrow[r] & \text{after} \arrow[r] & {} \\
& & & \text{finishedBy} \arrow[r] & \text{contains} \arrow[r] & \text{startedBy} \arrow[u]
\end{tikzcd}$$

For instance "$\boldsymbol{a}$ overlaps $\boldsymbol{b}$"—equivalently $\boldsymbol{a} \circledcirc \boldsymbol{b}$ has the value `overlaps`—is the case $\underline{a} < \underline{b} < \overline{a} < \overline{b}$.

The three extra values are: `bothEmpty` when $\boldsymbol{a} = \boldsymbol{b} = \emptyset$, else `firstEmpty` when $\boldsymbol{a} = \emptyset$, `secondEmpty` when $\boldsymbol{b} = \emptyset$.

Table 6 shows the 16 states, with the 13 "nonempty" states specified (a) in terms of set membership using quantifiers and (b) in terms of the endpoints $\underline{a}, \overline{a}, \underline{b}, \overline{b}$, and also (c) shown diagrammatically.

The set and endpoint specifications remove some ambiguities of the diagram view when one interval shrinks to a single point that coincides with an endpoint of the other. Such a case is allocated to `equal` when all four endpoints coincide; else to `starts`, `finishes`, `finishedBy` or `startedBy` as appropriate; never to `meets` or `metBy`.

[*Note. The 16 state values can be encoded in four bits. However, if they are then translated into patterns P in a 16-bit word, having one position equal to 1 and the rest zero, one can easily implement interval comparisons by using bit-masks.*

For instance, suppose we make the states s in Table 6's order correspond to the 16 bits in the word, left-to-right, so $s =$ `bothEmpty` *maps to $P(s) =$ 1000000000000000, $s =$* `firstEmpty` *maps to $P(s) =$ 0100000000000000 and so on. Consider the relation* `areDisjoint`$(\boldsymbol{a}, \boldsymbol{b})$*. This is true if and only if one or both of $\boldsymbol{a}$ or $\boldsymbol{b}$ is empty, or $\boldsymbol{a}$ is "before" $\boldsymbol{b}$, or $\boldsymbol{a}$ is "after" $\boldsymbol{b}$. That is, iff the logical "and" of $P(s)$ with the mask* `disjointMask = 1111000000000001` *is not identically zero.*

This scheme can be efficiently implemented in hardware, see for instance M. Nehmeier, S. Siegel and J. Wolff von Gudenberg [**6**]*. All the required comparisons in this standard can be implemented in this way, as can be, e.g., the "possibly" and "certainly" comparisons of Sun's interval Fortran. Thus the overlap operation is a primitive from which it is simple to derive all interval comparisons commonly found in the literature.*]

9.7.3. *Slope functions.*

The functions in Table 7 are the commonest ones needed to efficiently implement improved range enclosures via *first- and second-order slope* algorithms. They are analytic at $x = 0$ after filling in the removable singularity there, where each has the value 1.

TABLE 6. The 16 states of interval overlapping situations for intervals $\boldsymbol{a}, \boldsymbol{b}$. Notation $\forall_a$ means "for all a in $\boldsymbol{a}$", and so on. Phrases within a cell are joined by "and", e.g. `starts` is specified by $(\underline{a} = \underline{b} \wedge \overline{a} < \overline{b})$.

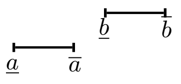
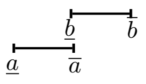
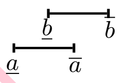
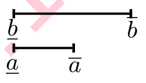
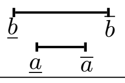
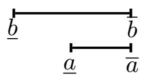
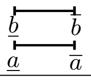
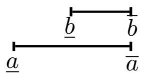
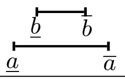
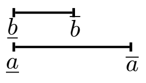
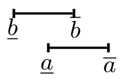
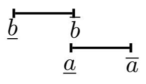
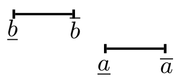
State $\boldsymbol{a} \circledcirc \boldsymbol{b}$ is	Set specification	Endpoint specification	Diagram
States with either interval empty			
`bothEmpty`	$\boldsymbol{a} = \emptyset \wedge \boldsymbol{b} = \emptyset$		
`firstEmpty`	$\boldsymbol{a} = \emptyset \wedge \boldsymbol{b} \neq \emptyset$		
`secondEmpty`	$\boldsymbol{a} \neq \emptyset \wedge \boldsymbol{b} = \emptyset$		
States with both intervals nonempty			
`before`	$\forall_a \forall_b\ a < b$	$\overline{a} < \underline{b}$	 $\underline{b}$ $\overline{b}$ $\underline{a}$ $\overline{a}$
`meets`	$\forall_a \forall_b\ a \leq b$ $\exists_a \forall_b\ a < b$ $\exists_a \exists_b\ a = b$	$\underline{a} < \overline{a}$ $\overline{a} = \underline{b}$ $\underline{b} < \overline{b}$	 $\underline{b}$ $\overline{b}$ $\underline{a}$ $\overline{a}$
`overlaps`	$\exists_a \forall_b\ a < b$ $\exists_b \forall_a\ a < b$ $\exists_a \exists_b\ b < a$	$\underline{a} < \underline{b}$ $\underline{b} < \overline{a}$ $\overline{a} < \overline{b}$	 $\underline{b}$ $\overline{b}$ $\underline{a}$ $\overline{a}$
`starts`	$\exists_a \forall_b\ a \leq b$ $\exists_b \forall_a\ b \leq a$ $\exists_b \forall_a\ a < b$	$\underline{a} = \underline{b}$ $\overline{a} < \overline{b}$	 $\underline{b}$ $\overline{b}$ $\underline{a}$ $\overline{a}$
`containedBy`	$\exists_b \forall_a\ b < a$ $\exists_b \forall_a\ a < b$	$\underline{b} < \underline{a}$ $\overline{a} < \overline{b}$	 $\underline{b}$ $\overline{b}$ $\underline{a}$ $\overline{a}$
`finishes`	$\exists_b \forall_a\ b < a$ $\exists_a \forall_b\ b \leq a$ $\exists_b \forall_a\ a \leq b$	$\underline{b} < \underline{a}$ $\overline{a} = \overline{b}$	 $\underline{b}$ $\overline{b}$ $\underline{a}$ $\overline{a}$
`equal`	$\forall_a \exists_b\ a = b$ $\forall_b \exists_a\ b = a$	$\underline{a} = \underline{b}$ $\overline{a} = \overline{b}$	 $\underline{b}$ $\overline{b}$ $\underline{a}$ $\overline{a}$
`finishedBy`	$\exists_a \forall_b\ a < b$ $\exists_b \forall_a\ a \leq b$ $\exists_a \forall_b\ b \leq a$	$\underline{a} < \underline{b}$ $\overline{b} = \overline{a}$	 $\underline{b}$ $\overline{b}$ $\underline{a}$ $\overline{a}$
`contains`	$\exists_a \forall_b\ a < b$ $\exists_a \forall_b\ b < a$	$\underline{a} < \underline{b}$ $\overline{b} < \overline{a}$	 $\underline{b}$ $\overline{b}$ $\underline{a}$ $\overline{a}$
`startedBy`	$\exists_b \forall_a\ b \leq a$ $\exists_a \forall_b\ a \leq b$ $\exists_a \forall_b\ b < a$	$\underline{b} = \underline{a}$ $\overline{b} < \overline{a}$	 $\underline{b}$ $\overline{b}$ $\underline{a}$ $\overline{a}$
`overlappedBy`	$\exists_b \forall_a\ b < a$ $\exists_a \forall_b\ b < a$ $\exists_b \exists_a\ a < b$	$\underline{b} < \underline{a}$ $\underline{a} < \overline{b}$ $\overline{b} < \overline{a}$	 $\underline{b}$ $\overline{b}$ $\underline{a}$ $\overline{a}$
`metBy`	$\forall_b \forall_a\ b \leq a$ $\exists_b \exists_a\ b = a$ $\exists_b \forall_a\ b < a$	$\underline{b} < \overline{b}$ $\overline{b} = \underline{a}$ $\underline{a} < \overline{a}$	 $\underline{b}$ $\overline{b}$ $\underline{a}$ $\overline{a}$
`after`	$\forall_b \forall_a\ b < a$	$\overline{b} < \underline{a}$	 $\underline{b}$ $\overline{b}$ $\underline{a}$ $\overline{a}$

TABLE 7. Recommended slope functions.

Name	Definition	Point function domain	Point function range	Note
`expSlope1`(x)	$\frac{1}{x}(e^x - 1)$	$\mathbb{R}$	$(0, \infty)$	
`expSlope2`(x)	$\frac{2}{x^2}(e^x - 1 - x)$	$\mathbb{R}$	$(0, \infty)$	
`logSlope1`(x)	$\frac{2}{x^2}(\log(1+x) - x)$	$\mathbb{R}$	$(0, \infty)$	
`logSlope2`(x)	$\frac{3}{x^3}(\log(1+x)-x+\frac{x^2}{2})$	$\mathbb{R}$	$(0, \infty)$	
`cosSlope2`(x)	$-\frac{2}{x^2}(\cos x - 1)$	$\mathbb{R}$	$[0, 1]$	
`sinSlope3`(x)	$-\frac{6}{x^3}(\sin x - x)$	$\mathbb{R}$	$(0, 1]$	
`asinSlope3`(x)	$\frac{6}{x^3}(\arcsin x - x)$	$[-1, 1]$	$[1, 3\pi - 6]$	
`atanSlope3`(x)	$-\frac{3}{x^3}(\arctan x - x)$	$\mathbb{R}$	$(0, 1]$	
`coshSlope2`(x)	$\frac{2}{x^2}(\cosh x - 1)$	$\mathbb{R}$	$[1, \infty)$	
`sinhSlope3`(x)	$\frac{3}{x^3}(\sinh x - x)$	$\mathbb{R}$	$[\frac{1}{2}, \infty)$	

10. The decoration system at Level 1

10.1. Decorations and decorated intervals overview. The decoration system of the set-based flavor conforms to the principles of Clause 8.1. An implementation makes the decoration system available by providing:

– a decorated version of each interval extension of an arithmetic operation, of each interval constructor, and of some other operations;
– various auxiliary functions, e.g., to extract a decorated interval's interval and decoration parts, and to apply a standard initial decoration to an interval.

The system is specified here at a mathematical level, with the finite-precision aspects in §12. §10.2, 10.3, 10.4 give the basic concepts. §10.5, 10.6 define how intervals are given an initial decoration, and how decorations are bound to library interval arithmetic operations to give correct propagation through expressions. §10.7, 10.8 are about non-arithmetic operations. §10.9 describes housekeeping operations on decorations, including comparisons, and conversion between a decorated interval and its interval and decoration parts. §10.10 discusses the decoration of user-defined arithmetic operations. The decoration `com` makes it possible to verify, under fairly restrictive conditions, whether a given computation gives the same result in different flavors; §10.11 gives explanatory notes on the `com` decoration. §10.12 defines a restricted decorated interval arithmetic that suffices for some important applications and is easier to implement efficiently.

In Annex D, Clause 19 gives examples of the meaning and use of decorations; and Clause 22 contains a rigorous theoretical foundation, including a proof of the Fundamental Theorem of Decorated Interval Arithmetic for this flavor.

10.2. Definitions and basic properties. Formally, a decoration d is a property (that is, a boolean-valued function) $p_d(f, \boldsymbol{x})$ of pairs $(f, \boldsymbol{x})$, where f is a real-valued function with domain $\mathrm{Dom}(f) \subseteq \mathbb{R}^n$ for some $n \geq 0$ and $\boldsymbol{x} \in \overline{\mathbb{IR}}^n$ is an n-dimensional box, regarded as a subset of $\mathbb{R}^n$. The notation $(f, \boldsymbol{x})$ unless said otherwise denotes such a pair, for arbitrary n, f and $\boldsymbol{x}$. Equivalently, d is identified with the set of pairs for which the property holds:

$$d = \{\, (f, \boldsymbol{x}) \mid p_d(f, \boldsymbol{x}) \text{ is true} \,\}. \tag{16}$$

The set $\mathbb{D}$ of decorations has five members:

Value	Short description	Property	Definition
`com`	common	$p_{\texttt{com}}(f, \boldsymbol{x})$	$\boldsymbol{x}$ is a bounded, nonempty subset of $\mathrm{Dom}(f)$; f is continuous at each point of $\boldsymbol{x}$; and the computed interval $f(\boldsymbol{x})$ is bounded.
`dac`	defined & continuous	$p_{\texttt{dac}}(f, x)$	$\boldsymbol{x}$ is a nonempty subset of $\mathrm{Dom}(f)$, and the restriction of f to $\boldsymbol{x}$ is continuous;
`def`	defined	$p_{\texttt{def}}(f, \boldsymbol{x})$	$\boldsymbol{x}$ is a nonempty subset of $\mathrm{Dom}(f)$;
`trv`	trivial	$p_{\texttt{trv}}(f, \boldsymbol{x})$	always true (so gives no information);
`ill`	ill-formed	$p_{\texttt{ill}}(f, \boldsymbol{x})$	Not an Interval; formally $\mathrm{Dom}(f) = \emptyset$, see §10.3.

(17)

These are listed according to the propagation order (27), which may also be thought of as a quality-order of $(f, \boldsymbol{x})$ pairs—decorations above `trv` are "good" and those below are "bad".

A **decorated interval** is a pair, written interchangeably as $(\boldsymbol{u}, d)$ or $\boldsymbol{u}_d$, where $\boldsymbol{u} \in \overline{\mathbb{IR}}$ is a real interval and $d \in \mathbb{D}$ is a decoration. $(\boldsymbol{u}, d)$ may also denote a decorated box $\big((\boldsymbol{u}_1, d_1), \ldots, (\boldsymbol{u}_n, d_n)\big)$, where $\boldsymbol{u}$ and d are the vectors of interval parts $\boldsymbol{u}_i$ and decoration parts d_i, respectively. The set of decorated intervals is denoted by $\overline{\mathbb{DIR}}$, and the set of decorated boxes with n components is denoted by $\overline{\mathbb{DIR}}^n$.

When several named intervals are involved, the decorations attached to $\boldsymbol{u}, \boldsymbol{v}, \ldots$ are often named $du, dv, \ldots$ for readability, for instance $(\boldsymbol{u}, du)$ or $\boldsymbol{u}_{du}$, etc.

An interval or decoration may be called a **bare** interval or decoration, to emphasize that it is not a decorated interval.

Treating the decorations as sets as in (16), `trv` is the set of all $(f, \boldsymbol{x})$ pairs, and the others are nonempty subsets of `trv`. By design they satisfy the **exclusivity rule**

For any two decorations, either one contains the other or they are disjoint. (18)

Namely the definitions (17) give:

$$\texttt{com} \subset \texttt{dac} \subset \texttt{def} \subset \texttt{trv} \supset \texttt{ill}, \qquad \text{note the change from } \subset \text{ to } \supset; \tag{19}$$

$$\texttt{com}, \texttt{dac} \text{ and } \texttt{def} \text{ are disjoint from } \texttt{ill}. \tag{20}$$

Property (18) implies that for any $(f, \boldsymbol{x})$ there is a unique tightest (in the containment order (19)), decoration such that $p_d(f, \boldsymbol{x})$ is true, called the **strongest decoration of** $(f, \boldsymbol{x})$, or of f over $\boldsymbol{x}$, and written $\operatorname{dec}(f, \boldsymbol{x})$. That is:

$$\operatorname{dec}(f, \boldsymbol{x}) = d \iff p_d(f, \boldsymbol{x}) \text{ holds, but } p_e(f, \boldsymbol{x}) \text{ fails for all } e \subset d. \tag{21}$$

[*Note. Like the exact range* $\operatorname{Rge}(f \mid \boldsymbol{x})$*, the strongest decoration is theoretically well-defined, but its value for a particular* f *and* $\boldsymbol{x}$ *may be impractically expensive to compute, or even undecidable.*]

10.3. The ill-formed interval. The `ill` decoration results from invalid constructions, and propagates unconditionally through arithmetic expressions. Namely, if a constructor call does not return a valid decorated interval, it returns an ill-formed one (i.e., decorated with `ill`); and the decorated interval result of a library arithmetic operation is ill-formed, if and only if one of its inputs is ill-formed. Formally, `ill` may be identified with the property $\operatorname{Dom}(f) = \emptyset$ of $(f, \boldsymbol{x})$ pairs, see Clause 24.

An ill-formed decorated interval is also called NaI, **Not an Interval**. Except as described in the next paragraph, an implementation shall behave as if there is only one NaI, whose interval part is indeterminate. However, the `intervalPart()` operation must return a bare interval for any decorated interval input, and for NaI this shall be Empty; thus NaI may be viewed as being $\emptyset_{\texttt{ill}}$.

An exception is that other information may be stored in an NaI in an implementation-defined way (like the payload of a 754 floating point NaN), and functions may be provided for a user to set and read this for diagnostic purposes. An implementation may also provide means for an exception to be raised when an NaI is produced.

[*Example. The constructor call* `nums2interval`$(2, 1)$ *is invalid in this flavor, so its decorated version returns* NaI.]

10.4. Permitted combinations. A decorated interval $\boldsymbol{y}_{dy}$ shall always be such that $\boldsymbol{y} \supseteq \operatorname{Rge}(f \mid \boldsymbol{x})$ and $p_{dy}(f, \boldsymbol{x})$ holds, for some $(f, \boldsymbol{x})$ as in §10.2—informally, it must tell the truth about some conceivable evaluation of a function over a box. If dy = `dac` or `def` then by definition $\boldsymbol{x}$ is nonempty, and f is everywhere defined on it, so that $\operatorname{Rge}(f \mid \boldsymbol{x})$ is nonempty, implying $\boldsymbol{y}$ is nonempty. Hence the decorated intervals $\emptyset_{\texttt{dac}}$ and $\emptyset_{\texttt{def}}$, and $\emptyset_{\texttt{com}}$ if `com` is provided, are contradictory: implementations shall not produce them.

No other combinations are essentially forbidden.

10.5. Initial decoration. Correct use of decorations when evaluating an expression has two parts: correctly initialize the input intervals; and evaluate using decorated interval extensions of library operations. To provide correct initialization, the function `newDec()` is provided. For a single bare interval $\boldsymbol{x}$, `newDec()` decorates it with $\operatorname{dec}(\operatorname{Id}, \boldsymbol{x})$, the strongest decoration dx that makes $p_{dx}(\operatorname{Id}, \boldsymbol{x})$ true, where Id is the identity function $\operatorname{Id}(x) = x$ for real x. That is,

$$\texttt{newDec}(\boldsymbol{x}) = \boldsymbol{x}_d \quad \text{where} \quad d = \operatorname{dec}(\operatorname{Id}, \boldsymbol{x}) = \begin{cases} \texttt{com} & \text{if } \boldsymbol{x} \text{ is nonempty and bounded,} \\ \texttt{dac} & \text{if } \boldsymbol{x} \text{ is unbounded,} \\ \texttt{trv} & \text{if } \boldsymbol{x} \text{ is empty.} \end{cases} \tag{22}$$

[*Note. The following features are used in examples in this document. It is language-defined whether an implementation provides them.*

– *For a decorated interval* $\boldsymbol{x}_{dx}$, `newDec()` *discards the decoration and acts on the interval part,*

$$\texttt{newDec}(\boldsymbol{x}_{dx}) = \texttt{newDec}(\boldsymbol{x}). \tag{23}$$

– *For a vector of* n *bare or decorated intervals,* `newDec()` *acts componentwise to give a vector of* n *decorated intervals.*

– *A bare interval constant, in a context that expects a decorated interval, is implicitly* promoted *to a decorated interval by applying the* `newDec` *function. For example,* $[1, 2]$, $[1, +\infty]$ *and* $\emptyset$ *are promoted to* $[1, 2]_{\texttt{com}}$, $[1, +\infty]_{\texttt{dac}}$ *and* $\emptyset_{\texttt{trv}}$ *respectively.*

]

10.6. Decorations and arithmetic operations. Given a scalar point function φ of k variables, a **decorated interval extension** of φ—denoted here by the same name φ—adds a decoration component to a bare interval extension of φ. It has the form $\boldsymbol{w}_{dw} = \varphi(\boldsymbol{v}_{dv})$, where $\boldsymbol{v}_{dv} = (\boldsymbol{v}, dv)$ is a k-component decorated box $((\boldsymbol{v}_1, dv_1), \ldots, (\boldsymbol{v}_k, dv_k))$. By the definition of a bare interval extension, the interval part $\boldsymbol{w}$ depends only on the input intervals $\boldsymbol{v}$; the decoration part dw generally depends on both $\boldsymbol{v}$ and dv. In this context, NaI is regarded as being $\emptyset_{\texttt{ill}}$.

The definition of a bare interval extension implies

$$\boldsymbol{w} \supseteq \mathrm{Rge}(\varphi \,|\, \boldsymbol{v}), \qquad \text{(enclosure)}. \tag{24}$$

The decorated interval extension of φ determines a dv_0 such that

$$p_{dv_0}(\varphi, \boldsymbol{v}) \text{ holds}, \qquad \text{(a "local decoration")}. \tag{25}$$

It then evaluates the output decoration dw by

$$dw = \min\{dv_0, dv_1, \ldots, dv_k\}, \qquad \text{(the "min-rule")}, \tag{26}$$

where the minimum is taken with respect to the **propagation order**:

$$\texttt{com} > \texttt{dac} > \texttt{def} > \texttt{trv} > \texttt{ill}. \tag{27}$$

[*Notes.*

1. *Because* NaI *is treated as* $\emptyset_{\texttt{ill}}$*, this definition implies (without treating it as a special case) that* $\varphi(\boldsymbol{v}_{dv})$ *is* NaI *if, and only if, some component of* $\boldsymbol{v}_{dv}$ *is* NaI*.*
2. *Let* $f(z_1, \ldots, z_n)$ *be an expression defining a real point function* $f(x_1, \ldots, x_n)$*. Then decorated interval evaluation of* f *on a correctly initialized input decorated box* $\boldsymbol{x}_{dx}$ *gives a decorated interval* $\boldsymbol{y}_{dy}$ *such that not only, by the Fundamental Theorem of Interval Arithmetic, one has*

$$\boldsymbol{y} \supseteq \mathsf{Rge}(f \,|\, \boldsymbol{x}) \tag{28}$$

but also

$$p_{dy}(f, \boldsymbol{x}) \textit{ holds.} \tag{29}$$

For instance, if the computed dy *equals* `def` *then* f *is proven to be everywhere defined on the box* $\boldsymbol{x}$*. This is the* **Fundamental Theorem of Decorated Interval Arithmetic** *(FTDIA). The rules for initializing and propagating decorations are key to its validity. They are justified, and a formal statement and proof of the FTDIA given, in Annex D.*

Briefly, (22) gives the correct result for the simplest expression of all, where f *is the identity* $f(x) = x$*, which contains no arithmetic operations. The decorations are designed so that the min-rule (26) embodies basic facts of set theory and analysis, such as "If each of a set of functions is everywhere defined [resp. continuous] on its input, their composition has the same property" and "If any of a set of functions is nowhere defined on its input, their composition has the same property". It causes correct propagation of decorations through each arithmetic operation, and hence through a whole expression.*

3. *In the same way as the enclosure requirement (24) is compatible with many bare interval extensions, typically coming from different interval types at Level 2, so there may be several* dv_0 *satisfying the local decoration requirement (25). The ideal choice is the strongest decoration* d *such that* $p_d(\varphi, \boldsymbol{v})$ *holds, that is to take*

$$dv_0 = \mathsf{dec}(\varphi, \boldsymbol{v}). \tag{30}$$

This is easily computable in finite precision for the arithmetic operations in §9.6, 9.7—see the tables in Annex D, §18. However, functions may be added to the library in future for which (30) is impractical to compute for some arguments $\boldsymbol{v}$*. Hence the weaker requirement (25) is made.*

]

10.7. Decorations and non-arithmetic operations.

The following give interval results but are not interval extensions of point functions:

– the reverse-mode operations of §9.6.5;
– the cancellative operations $\texttt{cancelPlus}(\boldsymbol{x}, \boldsymbol{y})$ and $\texttt{cancelMinus}(\boldsymbol{x}, \boldsymbol{y})$ of §9.6.6;
– The set-oriented operations $\texttt{intersection}(\boldsymbol{x}, \boldsymbol{y})$ and $\texttt{convexHull}(\boldsymbol{x}, \boldsymbol{y})$ of §9.6.7.

The decorated interval version of each such operation distinguishes NaI inputs but otherwise returns a decoration indicating no information, as follows. If any input is NaI, the result is NaI. Otherwise the result is obtained by applying the corresponding operation to the interval parts of the inputs, and decorating its result with `trv`.

The user is responsible for applying a more informative decoration to the result via `setDec`(), where this can be deduced in a given context. Other versions of the operations may be provided in a language- or implementation-defined way, that apply such a context-adapted decoration.

The following operations give non-interval results:

– the numeric functions of §9.6.9;
– the boolean-valued functions of §9.6.10;
– the overlap function of §9.7.2.

For each such operation, if any input is NaI the result is undefined at Level 1. Otherwise, the operation acts on decorated intervals by discarding the decoration and applying the corresponding bare interval operation.

10.8. Boolean functions of decorated intervals. The equality comparison `equal`, or $=$, shall be provided for decorated intervals. NaI shall compare unequal to any decorated interval, including itself. Other input combinations shall compare equal if and only if the interval parts are equal and the decoration parts are equal.

The inequality comparison `notEqual`, or $\neq$, shall be provided. It is the logical negation of $=$ (so NaI $\neq$ NaI is true).

The unary function `isNaI` shall be provided. It is true if and only if its input is NaI.

10.9. Operations on decorations.

Given a decorated interval $\boldsymbol{x}_{dx}$, the operations `intervalPart`$(\boldsymbol{x}_{dx})$ and `decorationPart`$(\boldsymbol{x}_{dx})$ shall be provided, which return $\boldsymbol{x}$ and dx respectively. NaI is deemed to equal $\emptyset_{\texttt{ill}}$, so that `intervalPart`(NaI) returns $\emptyset$ and `decorationPart`(NaI) returns `ill`.

Given an interval $\boldsymbol{x}$ and a decoration dx, the operation `setDec`$(\boldsymbol{x}, dx)$ returns the decorated interval $\boldsymbol{x}_{dx}$. If this would produce one of the forbidden combinations—that is, $\boldsymbol{x} = \emptyset$ and dx is one of `def`, `dac` or `com`—then NaI is returned instead. `setDec`$(\boldsymbol{x},$ `ill`$)$ returns NaI for any interval $\boldsymbol{x}$, whether empty or not.

For decorations, comparison operations for equality $=$ and its negation $\neq$ shall be provided, as well as comparisons $>, <, \geq, \leq$ with respect to the propagation order (27).

[*Note. Careless use of the* `setDec` *function can negate the aims of the decoration system and lead to false conclusions that violate the FTDIA. It is provided for expert users, who may need it, e.g., to decorate the output of functions whose definition involves the* `intersection` *and* `convexHull` *operations.*]

10.10. User-supplied functions. A user may define a decorated interval extension of some point function, as defined in §10.6, to be used within expressions as if it were a library operation.

[*Examples.*

(i) In an application, an interval extension of the function

$$f(x) = x + 1/x$$

was required. Evaluated as written, it gives unnecessarily pessimistic enclosures: e.g., with $\boldsymbol{x} = [\frac{1}{2}, 2]$, *one obtains*

$$f(\boldsymbol{x}) = [\tfrac{1}{2}, 2] + 1/[\tfrac{1}{2}, 2] = [\tfrac{1}{2}, 2] + [\tfrac{1}{2}, 2] = [1, 4],$$

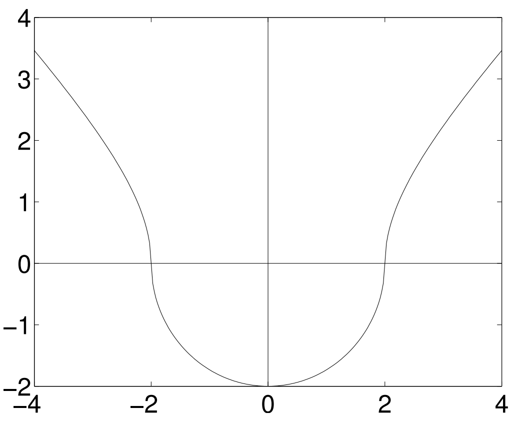
much wider than $\mathrm{Rge}(f \mid \boldsymbol{x}) = [2, 2\frac{1}{2}]$.

Thus it is useful to code a tight interval extension by special methods, e.g. monotonicity arguments, and provide this as a new library function. Suppose this has been done. To convert it to a decorated interval extension just entails adding code to provide a local decoration and combine this with the input decoration by the min-rule (26). In this case it is straightforward to compute the strongest local decoration $d = \mathrm{dec}(f, \boldsymbol{x})$, *as follows.*

$$d = \begin{cases} \texttt{com} & \textit{if } 0 \notin \boldsymbol{x} \textit{ and } \boldsymbol{x} \textit{ is nonempty and bounded,} \\ \texttt{dac} & \textit{if } 0 \notin \boldsymbol{x} \textit{ and } \boldsymbol{x} \textit{ is unbounded,} \\ \texttt{trv} & \textit{if } \boldsymbol{x} = \emptyset \textit{ or } 0 \in \boldsymbol{x} \neq [0, 0]. \end{cases}$$

(ii)

The next example shows how an expert may manipulate decorations explicitly to give a function, defined piecewise by different formulas in different regions of its domain, the best possible decoration. Suppose that



$$f(x) = \begin{cases} f_1(x) := \sqrt{x^2-4} & \text{if } |x|>2, \\ f_2(x) := -\sqrt{4-x^2} & \text{otherwise,} \end{cases}$$

where $:=$ means "defined as", see the diagram.

The function consists of three pieces, on regions $x \le -2$, $-2 \le x \le 2$ and $x \ge 2$, that join continuously at region boundaries, but the standard gives no way to determine this continuity, at run time or otherwise. For instance, if f is implemented by the case *function, the continuity information is lost when evaluating it on, say, $\boldsymbol{x} = [1,3]$, where both branches contribute for different values of $x \in \boldsymbol{x}$.*

However, a user-defined decorated interval function as defined below provides the best possible decorations.

```
function $\boldsymbol{y}_{dy} = f(\boldsymbol{x}_{dx})$
$\boldsymbol{u} = f_1(\boldsymbol{x} \cap [-\infty,-2])$
$\boldsymbol{v} = f_2(\boldsymbol{x} \cap [-2,2])$
$\boldsymbol{w} = f_1(\boldsymbol{x} \cap [2,+\infty])$
$\boldsymbol{y} = \texttt{convexHull}(\texttt{convexHull}(\boldsymbol{u},\boldsymbol{v}),\boldsymbol{w})$
$dy = dx$
```

The user's knowledge that f is everywhere defined and continuous is expressed by the statement $dy = dx$, propagating the input decoration unchanged. f, thus defined, can safely be used within a larger decorated interval evaluation.

]

10.11. Notes on the com decoration.

[*Notes.*

– *The force of* com *is the Level 2 property that the* computed *interval $f(\boldsymbol{x})$ is bounded. Equivalently, overflow did not occur, where overflow has the generalized meaning that a finite-precision operation could not enclose a mathematically bounded result in a bounded interval of the required output type. Briefly, for a single operation, "*com *is* dac *plus bounded inputs and no overflow".*

Thus the result of interval-evaluating an arithmetic expression in finite precision is decorated com *if and only if the evaluation is* common at Level 2, *meaning: each input that affects the result is nonempty and bounded, and each individual operation that affects the result is everywhere defined and continuous on its inputs and does not overflow.*

– *A tempting alternative is to make* com *record whether the evaluation is* common at Level 1, *meaning that all the relevant intervals are mathematically bounded, even if overflow occurred in finite precision. E.g., one might drop the "bounded inputs" requirement and require "mathematically bounded" instead of "actually bounded" on the output of an operation.*

However, the dac *decoration already provides such information and the suggested change gives nothing extra. Namely, if the inputs $\boldsymbol{x}$ to $f(\boldsymbol{x})$ are bounded, and the output decoration is* dac, *it follows, from the fact that a continuous function on a compact set is bounded, that the point function f is mathematically bounded on $\boldsymbol{x}$, and all its individual operations are mathematically bounded on their inputs even if overflow may have occurred in finite precision.*

For example consider $f(x) = 1/(2x)$ evaluated at $\boldsymbol{x} = [1, M]$ using an inf-sup type where M is the largest representable real. This gives

$$\boldsymbol{y}_{dy} = f(\boldsymbol{x}_{\texttt{com}}) = 1/(2 * [1, M]_{\texttt{com}}) = 1/[2, +\infty]_{\texttt{dac}} = [0, \tfrac{1}{2}]_{\texttt{dac}}.$$

Despite the overflow, one can deduce from the final dac *that the result of the multiplication was mathematically bounded.*

This may be of limited use: consider $g(x) = 1/f(x) = 1/(1/(2x))$, evaluated at the same $\boldsymbol{x} = [1, M]$ giving $\boldsymbol{z}_{dz}$. The standard has no way to record that the lower bound of $\boldsymbol{y}$ is mathematically positive, i.e., $1/(2M)$. Thus the Level 2 result is $\boldsymbol{z}_{dz} = [2, +\infty]_{\texttt{trv}}$, compare $[2, 2M]_{\texttt{com}}$ at Level 1.
]

10.12. Compressed arithmetic with a threshold (optional).

10.12.1. *Motivation.* The **compressed decorated interval arithmetic** (compressed arithmetic for short) described here lets experienced users obtain more efficient execution in applications where the use of decorations is limited to the context described below. An implementation need not provide it; if it does so, the behavior described in this subclause is required.

Each Level 2 instance of compressed arithmetic is based on a supported Level 2 bare interval type $\mathbb{T}$, but is a distinct "compressed type", with its own datums and library of operations.

The context is that of evaluating an arithmetic expression, where the use made of a decorated interval evaluation $\boldsymbol{y}_{dy} = f(\boldsymbol{x}_{dx})$ depends on a check of the result decoration dy against an application-dependent **exception threshold** τ, where $\tau \geq \texttt{trv}$ in the propagation order (27):

$dy \geq \tau$ represents normal computation. The decoration is not used, but one exploits the range enclosure given by the interval part and the knowledge that dy remained $\geq \tau$.

$dy < \tau$ declares an exception to have occurred. The interval part is not used, but one exploits the information given by the decoration.

10.12.2. *Compressed interval types.* For such uses, one needs to record an interval's value, or its decoration, but never both at once. The **compressed type** of threshold τ, **associated with** $\mathbb{T}$, is the type each of whose datums is either a bare $\mathbb{T}$-interval or a bare decoration less than τ. It is denoted $\mathbb{T}_\tau$. Two such types are the same if and only if they have the same $\mathbb{T}$ and the same τ. A $\mathbb{T}_\tau$ datum can be any $\mathbb{T}$ datum or any decoration except that:

– Only decorations $< \tau$ occur; in particular `com` is never used.
– The empty interval $\emptyset$ is replaced by—equivalently, is regarded by the implementation as being—a new decoration `emp` added to the table in (17), whose defining property is

Value	Short description	Property	Definition
`emp`	empty	$p_{\texttt{emp}}(f, \boldsymbol{x})$	$\boldsymbol{x} \cap \mathrm{Dom}(f)$ is empty;

(31)

`emp` lies between `trv` and `ill` in the containment order (19) and the propagation order (27):

$$\texttt{com} \subset \texttt{dac} \subset \texttt{def} \subset \texttt{trv} \supset \texttt{emp} \supset \texttt{ill}, \tag{32}$$
$$\texttt{com} > \texttt{dac} > \texttt{def} > \texttt{trv} > \texttt{emp} > \texttt{ill}.$$

Since $\tau \geq \texttt{trv}$ it is always true that `emp` $< \tau$, which means that as soon as an empty result is produced while evaluating an expression, the $dy < \tau$ case has occurred.
[*Note. The reason for treating $\emptyset$ as a decoration $< \tau$ is that obtaining an empty result (e.g., by doing something like $\sqrt{[-2, -1]}$ while evaluating a function) is one of the "exceptions" that compressed interval computation should detect.*]

The only way to use compressed arithmetic with a threshold τ is to construct $\mathbb{T}_\tau$ datums. Conversion between compressed types, say from a $\mathbb{T}_\tau$-interval to a $\mathbb{T}'_{\tau'}$-interval, shall be equivalent to converting first to a normal decorated interval by `normalInterval`(), then between decorated interval types if $\mathbb{T} \neq \mathbb{T}'$, and finally to the output type by τ'-`compressedInterval`().

[*Note. Since, for any practical interval type $\mathbb{T}$, a decoration fits into less space than an interval, one can implement arithmetic on compressed interval datums that take up the same space as a bare interval of that type. For instance if $\mathbb{T}$ is the IEEE754* `binary64` *inf-sup type, a compressed interval uses 16 bytes, the same as a bare $\mathbb{T}$-interval; a full decorated $\mathbb{T}$-interval needs at least 17 bytes.*

Because compressed intervals must behave exactly like bare intervals as long as one does not fall below the threshold, and take up the same space, there is no room to encode τ as part of the interval's value.]

10.12.3. *Operations.* The enquiry function `isInterval`$(\boldsymbol{x})$ returns true if the compressed interval $\boldsymbol{x}$ is an interval, false if it is a decoration.

The constructor τ-`compressedInterval`() is provided for each threshold value τ. The result of τ-`compressedInterval`$(\boldsymbol{X})$, where $\boldsymbol{X} = (\boldsymbol{x}, dx)$ is a decorated $\mathbb{T}$-interval, is a $\mathbb{T}_\tau$-interval as follows:

```
if $dx \geq \tau$, return the $\mathbb{T}_\tau$-interval with value $\boldsymbol{x}$
else return the $\mathbb{T}_\tau$-interval with value $dx$.
```

τ-`compressedInterval`$(\boldsymbol{x})$ for a bare interval $\boldsymbol{x}$ is equivalent to τ-`compressedInterval`(`newDec`$(\boldsymbol{x})$).

The function `normalInterval`$(\boldsymbol{x})$ converts a $\mathbb{T}_\tau$-interval to a decorated interval of the parent type, as follows:

```
if $\boldsymbol{x}$ is an interval, return $(\boldsymbol{x}, \tau)$.
if $\boldsymbol{x}$ is a decoration $d$
  if $d$ is ill or emp, return $(\mathtt{Empty}, d)$
  else return $(\mathtt{Entire}, d)$.
```

Arithmetic operations on compressed intervals shall follow *worst case semantics* rules that treat a decoration in {`trv`, `def`, `dac`} as representing a set of decorated intervals, and are necessary if the fundamental theorem is to remain valid. Namely, inputs to each operation behave as follows:

– Operations purely on bare intervals are performed as if each $\boldsymbol{x}$ is the decorated interval $\boldsymbol{x}_\tau$, resulting in a decorated interval $\boldsymbol{y}_{dy}$ that is then converted back into a compressed interval. If $dy < \tau$, the result is the bare decoration dy, otherwise the bare interval $\boldsymbol{y}$.
– For operations with at least one bare decoration input, the result is always a bare decoration. A bare interval input is treated as in the previous item. A decoration d in {`emp`, `ill`} is treated as $\emptyset_d$. A decoration d in {`trv`, `def`, `dac`} is treated (conceptually) as $\boldsymbol{x}_d$ with an arbitrary nonempty interval $\boldsymbol{x}$. The decoration `com` cannot occur. Performing the resulting decorated interval operation on all such possible inputs leads to a set of all possible results $\boldsymbol{y}_{dy}$. The tightest decoration (in the containment order (32)) enclosing all resulting dy is returned.

As a result each operation returns an actual or implied decoration compatible with its input, so that in an extended evaluation, the final decoration using compressed arithmetic is never stronger than that produced by full decorated interval arithmetic.

⚠ (JDP, 2013.) I conjecture that the following may be an equivalent specification, but have not checked.

1. Each compressed interval argument is converted to a decorated interval by `normalInterval`;
2. the corresponding operation of the parent decorated interval type is performed;
3. the result, if an interval, is converted back to a compressed interval by τ-`compressedInterval`.

(33)

[*Example. Assuming* $\tau >$ `def`,
– *The division* `def`$/[1, 2]$ *becomes* $\boldsymbol{x}_{\mathtt{def}}/[1, 2]_\tau$ *with arbitrary nonempty interval* $\boldsymbol{x}$.
The result is always decorated `def`, *so returns* `def`.
– *But* $[1, 2]/$`def` *becomes* $[1, 2]_\tau/\boldsymbol{x}_{\mathtt{def}}$ *with arbitrary nonempty interval* $\boldsymbol{x}$.
The result can be decorated `def`, `trv` *or* `emp`, *so returns the tightest decoration containing these, namely* `trv`.
]

Since there are only a few decorations, one can prepare complete operation tables according to this rule, and only these tables need to be implemented. In Annex D, sample tables and worked examples are in Clause 20 and a proof of correctness of the compressed arithmetic system is in Clause 21.

If compressed arithmetic is implemented, it shall provide versions of all the required operations of §9.6, and it should provide the recommended operations of §9.7.

11. Level 2 description

11.1. Level 2 introduction. Objects and operations at Level 2 are said to have **finite precision**. They are the entities from which implementable interval algorithms may be constructed. Level 2 objects are called **datums**[1]Since the standard deals with numeric functions of intervals (such as the midpoint) and interval functions of numbers (such as the construction of an interval from its lower and upper bounds), this clause involves both numeric and interval datums, as well as the set $\mathbb{D}$ of decorations.

Following 754 terminology, numeric (floating point) datums are organized into **formats**. Interval datums are organized into **types**. Each format or type is a finite set of datums, with associated operations. The standard defines three kinds of interval type:

– **Bare interval types**, see §11.6, represent finite sets of (mathematical, Level 1) intervals.
– **Decorated interval types**, see §12, represent finite sets of decorated intervals.
– **Compressed interval types** (optional), see §12.5, implement compressed decorated interval arithmetic.

An implementation shall support at least one bare interval type. If 754-conforming, it shall support the inf-sup type, see §11.6.2, of at least one of the five basic formats of 754§3.3.

There shall be a one-to-one correspondence between bare interval types and decorated interval types, wherein each bare interval type has a corresponding *derived* decorated interval type, see §12.2. Beyond this, which types are supported is language- or implementation-defined.

This standard uses the term $\mathbb{T}$**-version** of an operation, where $\mathbb{T}$ is a bare or decorated interval type, to mean a finite-precision approximation to the corresponding Level 1 operation, in which any input or output intervals become $\mathbb{T}$-intervals. This includes the following:

(a) A $\mathbb{T}$-interval extension (§11.9) of one of the required or recommended arithmetic operations of §9.6, 9.7.
(b) A set operation, such as intersection and convex hull of $\mathbb{T}$-intervals, returning a $\mathbb{T}$-interval.
(c) A function such as the midpoint, whose input is a $\mathbb{T}$-interval and output is a numeric value.
(d) A constructor, whose input is numeric or text and output is a $\mathbb{T}$-interval.
(e) The $\mathbb{T}$-interval hull, regarded as a conversion operation, see §11.8.3.

Generically these comprise the **operations of the type** $\mathbb{T}$, for the implementation.

An implementation may also support mixed-type operations, where input and output intervals are not all of the same type.

It is language-defined whether the type of a datum can be determined at run time.

11.2. Naming conventions for operations. An operation can exist in many forms at Level 2 depending on the input and result types, and is generally given a name that suits the context.

For example, the addition of two interval datums $\boldsymbol{x}, \boldsymbol{y}$ may be written in generic algebra notation $\boldsymbol{x} + \boldsymbol{y}$; or with a generic text name `addition`$(\boldsymbol{x}, \boldsymbol{y})$; or giving full type information such as *decimal64-infsup*-`addition`$(\boldsymbol{x}, \boldsymbol{y})$.

It may also be written as $\mathbb{T}$-`addition`$(\boldsymbol{x}, \boldsymbol{y})$ to show it is an operation of a particular but unspecified type $\mathbb{T}$, or—in the context of 754-conforming types—as *typeOf*-`addition`$(\boldsymbol{x}, \boldsymbol{y})$ where *typeOf* has a similar meaning to 754's *formatOf*.

In a specific language or programming environment, the names used for types may differ from those used in this document.

11.3. 754-conformance. The standard defines the notion of 754-conformance, whose stronger requirements improve accuracy and programming convenience. In this context a part of an implementation means a subset of the set of supported Level 2 interval types.

A **754-conforming type** is an inf-sup type derived from a 754 floating point format—one of the five basic types or an extended precision or extendable precision format—that meets the general requirements for conformance and whose operations meet the accuracy requirements in Table 8.

A **754-conforming part** of an implementation is a subset of the set of supported 754-conforming types, that meets the requirements for mixed-type arithmetic in the next paragraphs.

[1]Not "data", whose common meaning could cause confusion.

A **754-conforming implementation** is one where *all* supported types are 754-conforming, and the whole set of supported types meets the requirements for mixed-type arithmetic.

11.3.1. *754-conforming mixed-type arithmetic.* The 754 standard requires a conforming floating point system to provide mixed-format "*formatOf*" operations. That is, the output format is specified and the inputs may be of any format of the same radix as the output. The result is computed as if using the exact inputs and rounded to the required accuracy on output. This eliminates the problem of double rounding in mixed-format work, which otherwise can cause significant growth of errors.

A 754-conforming part of an implementation shall provide (e.g., by exploiting the *formatOf* feature at Level 3) corresponding mixed-type "*typeOf*" operations. That is, the output type is specified and the inputs may be of any type of the same radix as the output. The result shall be computed as if using the exact inputs and shall meet the accuracy requirements for each operation, specified in Table 8.

These requirements shall apply to all Level 2 operations with interval operands, without explicit mention.

11.4. Tagging, and the meaning of equality at Level 2. A Level 2 format or type is an abstraction of a particular way to represent numbers or intervals—e.g., "IEEE 64 bit binary" for numbers—focusing on the Level 1 objects represented, and hiding the Level 3 method by which it is done.

However a datum is more than just the Level 1 value: for instance the number 3.75 represented in 64 bit binary is a different datum from the same number represented in 64 bit decimal.

This is achieved by formally regarding each datum as a pair:

$$\begin{aligned}\text{number datum} &= (\text{Level 1 number, format name}),\\ \text{interval datum} &= (\text{Level 1 interval, type name}),\end{aligned}$$

where the name is some symbol that uniquely identifies the format or type. The Level 1 value is said to be **tagged** by the name. It follows that distinct formats or types are disjoint sets

By convention, such names are omitted from datums except when clarity requires.

[*Example. Level 2 interval addition within a type named* t *is normally written* $\boldsymbol{z} = \boldsymbol{x} + \boldsymbol{y}$*, though the full correct form is* $(\boldsymbol{z}, t) = (\boldsymbol{x}, t) + (\boldsymbol{y}, t)$*. The full form might be used, for instance, to indicate that mixed-type addition is forbidden between types* s *and* t *but allowed between types* s *and* u*. Namely, one can say that* $(\boldsymbol{x}, s) + (\boldsymbol{y}, t)$ *is undefined, but* $(\boldsymbol{x}, s) + (\boldsymbol{y}, u)$ *is defined.*]

In this document the five basic formats of 754§3.3 are named `binary32`, `binary64`, `binary128`, `decimal64`, `decimal128`. Their inf-sup types (§11.6.2) are $\mathbb{I}$(`binary32`), ..., $\mathbb{I}$(`decimal128`), the parentheses being optional. Also, abbreviated names such as `b64` instead of `binary64` are sometimes used for convenience, and refer to the same format, or type derived from it.

An interval datum may have more than one representation at Level 3 (e.g., in an inf-sup type, a zero endpoint might be stored as either -0 or $+0$). Independent of representation, two bare interval datums are defined to be equal if and only if they represent the same Level 1 interval tagged by the same or equivalent name. In particular two datums of the same bare interval type are equal if and only if they represent the same Level 1 interval.

[*Note. Two bare interval datums* $\boldsymbol{a}, \boldsymbol{b}$ *of the same type are equal if and only if* `equal`$(\boldsymbol{a}, \boldsymbol{b})$ *returns* `true`*. However, in the context of 754-conforming types, intervals of different types can compare equal. E.g., let* $\boldsymbol{a}, \boldsymbol{b}$ *be the* `binary32` *inf-sup and* `binary64` *inf-sup extensions, respectively, of an interval that is represented exactly in both types, such as* $[1, 2]$*. Then* `equal`$(\boldsymbol{a}, \boldsymbol{b})$ *is* `true`*, but* $\boldsymbol{a}$ *and* $\boldsymbol{b}$ *are not equal in the sense of this subclause.*]

⚠ The next passage may be controversial, and certainly needs stating more precisely. Discussion please!

Implementations shall ensure that different Level 3 representations of an interval datum cannot affect the execution of a program by conforming to the

Equality principle. Let $y = f(x_1, x_2, ..., x_n)$ be an arbitrary evaluation at Level 2 of a predicate (boolean valued expression), where the arguments x_j can be any mix of intervals and numbers. Then y shall be unchanged if any *interval* argument x_j is replaced by an argument x'_j that equals x_j in the Level 2 sense.

Here f is in general a mixed interval/numeric expression. It is built up of library functions φ_i, which can be arithmetic operations, constructors, comparisons, or numeric functions of intervals, or point arithmetic operations or comparisons of the underlying floating point system.

Evaluation at Level 2 means that the inputs x_j are of specified types (using this to mean both interval types and number formats); and each φ_i has specified types for its inputs and output, such that each value that is used as an input is of the expected type.

[*Example. Let* $\mathbb{F}$ *be a 754 format and* $\mathbb{T} = \overline{\mathbb{IF}}$ *the derived inf-sup type. Suppose a* $\mathbb{T}$*-interval* $[l, u]$ *is represented at Level 3 as the pair of* $\mathbb{F}$*-numbers* (l, u)*. Let* f *be the expression*

$$1/\texttt{inf}(\boldsymbol{x}) > 0.$$

and consider $[0, 1]$ *with the two Level 3 representations* $\boldsymbol{x} = (-0, 1)$ *and* $\boldsymbol{x}' = (+0, 1)$*. Then* $\boldsymbol{x} = \boldsymbol{x}'$ *in the Level 2 sense, but a naive implementation gives*

$$f(\boldsymbol{x}) = \left(\frac{1}{\texttt{inf}(\boldsymbol{x})} > 0\right) \qquad = \left(\frac{1}{-0} > 0\right) \qquad = (-\infty > 0) = \texttt{false};$$
$$f(\boldsymbol{x}') = \left(\frac{1}{\texttt{inf}(\boldsymbol{x}')} > 0\right) \qquad = \left(\frac{1}{+0} > 0\right) \qquad = (+\infty > 0) = \texttt{true}.$$

The standard does not say which of these two results is "correct". But since they differ, the equality principle is violated and such an implementation is non-conforming. One way to make it conforming is to canonicalize all operations that output an interval, to ensure that for instance all zero bounds are stored as $+0$*.*]

11.5. Number formats. Having regard to §11.4, a **number format**, or just format, is the set of all pairs (x, f) where x belongs to a set $\mathbb{F}$, and f is a name for the format.

$\mathbb{F}$ comprises a finite subset of the extended reals $\overline{\mathbb{R}}$, together with a value NaN. A **numeric** member of $\mathbb{F}$ is one that is not NaN. $\mathbb{F}$ shall contain zero, $-\infty$ and $+\infty$, and shall be symmetric: if a numeric x is in $\mathbb{F}$, so is $-x$.

Following the convention of omitting names, the format is normally identified with the set $\mathbb{F}$, and one may say a number format is a set of datums comprising NaN together with a finite, symmetric, set $\mathbb{F}$ of extended reals that contains 0 and $\pm\infty$. The non-NaN members of $\mathbb{F}$ are called $\mathbb{F}$-numbers.

[*Note. At Level 2 each format has only one* NaN *datum, but this may correspond to more than one* NaN *at Level 3, e.g. by using the payload of a 754* NaN.]

A floating-point format in the 754 sense, such as `binary64`, is identified with the number format for which $\mathbb{F}$ is the set of extended-real numbers that are exactly representable in that format, where -0 and $+0$ both represent the mathematical number 0.

⚠ I deviate from motion 33 here in not allowing for $-0, +0$ to be distinct datums, hoping we can avoid this.

A number format $\mathbb{F}$ is said to be **compatible** with an interval type $\mathbb{T}$, if each non-empty $\mathbb{T}$-interval contains at least one finite $\mathbb{F}$-number.

Each format $\mathbb{F}$ shall have an associated **rounding function** that maps any extended real number x to an element of $\mathbb{F}$. A constraint can be given on the rounding direction, e.g. one of the rounding direction attributes in 754§4.3. The rounded result is an element of $\mathbb{F}$ that is closest to x subject to this constraint, with an implementation-defined rule for the distance to an infinity, and for the method of tie-breaking when more than one member of $\mathbb{F}$ has the "closest" property.

11.6. Bare interval types.

11.6.1. *Definition.* Having regard to §11.4, a **bare interval type** is the set of all pairs $(\boldsymbol{x}, t)$ where $\boldsymbol{x}$ belongs to a finite subset $\mathbb{T}$ of the mathematical intervals $\overline{\mathbb{IR}}$ that contains Empty and Entire, and t is a name for the type.

Following the convention of omitting names, the type is normally identified with the set $\mathbb{T}$, and one may say a bare interval type is an arbitrary finite set $\mathbb{T}$ of intervals that contains Empty and Entire.

A $\mathbb{T}$**-interval** means an interval belonging to $\mathbb{T}$; a $\mathbb{T}$**-box** means a box with $\mathbb{T}$-interval components.

[*Examples. To illustrate the flexibility allowed in defining types, let* S_1 *and* S_2 *be the sets of inf-sup intervals using 754 single (*`binary32`*) and double (*`binary64`*) precision respectively. That is, a* *member of S_1 [respectively S_2] is either empty, or an interval whose bounds are exactly representable in* `binary32` *[respectively* `binary64`*].*

An implementation can (and usually would) define these as different types, by tagging members of S_1 by one type name t_1 and members of S_2 by another name t_2. At Level 3 they would be represented as a pair of `binary32` *or* `binary64` *floating point datums respectively. However, it could treat them as one type, with the representation by a pair of* `binary32`*'s being a space-saving alternative to the pair of* `binary64`*'s, to be used when convenient.*]

11.6.2. *Inf-sup and mid-rad types.* The **inf-sup type derived from** a given number format $\mathbb{F}$ (the type $\mathbb{F}$ **inf-sup**, e.g., "`binary64` inf-sup") is the bare interval type $\mathbb{T}$ comprising all intervals whose endpoints are in $\mathbb{F}$, together with Empty. $\mathbb{F}$ is termed the **parent format** of $\mathbb{T}$. Note that Entire is in $\mathbb{T}$ because $\pm\infty \in \mathbb{F}$ by the definition of a number format, so $\mathbb{T}$ satisfies the requirements for a bare interval type given in §11.6.1.

Mid-rad types are not specified by this standard but are useful for examples. A mid-rad bare interval type is taken to be one whose nonempty bounded intervals comprise all intervals of the form $[m-r, m+r]$, where m is in some number format $\mathbb{F}$, and r is in a possibly different number format $\mathbb{F}'$, with m, r finite and $r \geq 0$. From the definition in §11.6.1 such a type, to be conforming, must contain Empty and Entire, so at Level 3 it must have representations of these; it may also have representations of semi-bounded intervals.

11.7. Multi-precision interval types. Multi-precision floating point systems—extendable precision in 754 terminology—generally provide an (at least conceptually) infinite sequence of levels of precision, where there is a finite set $\mathbb{F}_n$ of numbers representable at the nth level ($n = 1, 2, 3, \ldots$), and $\mathbb{F}_1 \subset \mathbb{F}_2 \subset \mathbb{F}_3 \ldots$. These are typically used to define a corresponding infinite sequence of interval types $\mathbb{T}_n$ with $\mathbb{T}_1 \subset \mathbb{T}_2 \subset \mathbb{T}_3 \ldots$.

[*Example. For multi-precision systems that define a nonempty $\mathbb{T}_n$-interval to be one whose endpoints are $\mathbb{F}_n$-numbers, each $\mathbb{T}_n$ is an inf-sup type with a unique interval hull operation—explicit, in the sense of §11.8.*]

A conforming implementation must define such $\mathbb{T}_n$ as a *parameterized sequence* of interval types. It cannot take the union over n of the sets $\mathbb{T}_n$ as a *single* type, because this infinite set has no interval hull operation: there is generally no tightest member of it enclosing a given set of real numbers. This constrains the design of conforming multi-precision interval systems.

11.8. Explicit and implicit types, and Level 2 hull operation.

11.8.1. *Hull in one dimension.* Each bare interval type $\mathbb{T}$ shall have an **interval hull** operation

$$\boldsymbol{y} = \mathrm{hull}_{\mathbb{T}}(\boldsymbol{s})$$

specified, which is part of its mathematical definition. It maps an arbitrary set of reals, $\boldsymbol{s}$, to a minimal $\mathbb{T}$-interval $\boldsymbol{y}$ enclosing $\boldsymbol{s}$. Minimal is in the sense that

$$\boldsymbol{s} \subseteq \boldsymbol{y}, \text{ and for any other } \mathbb{T}\text{-interval } \boldsymbol{z}, \text{ if } \boldsymbol{s} \subseteq \boldsymbol{z} \subseteq \boldsymbol{y} \text{ then } \boldsymbol{z} = \boldsymbol{y}.$$

For clarity when needed, this operation is called the $\mathbb{T}$-hull and denoted $\mathrm{hull}_{\mathbb{T}}$.

Since $\mathbb{T}$ is a finite set and contains Entire, such a minimal $\boldsymbol{y}$ exists for any $\boldsymbol{s}$. In general $\boldsymbol{y}$ may not be unique. If it is unique for every subset $\boldsymbol{s}$ of $\mathbb{R}$, then the type $\mathbb{T}$ is called **explicit**, otherwise it is **implicit**. For an explicit type, the hull operation is uniquely determined and need not be separately specified. For an implicit type, the implementation's documentation shall specify the hull operation, e.g., by an algorithm.

Two types with different hull operations are different, even if they have the same set of intervals.

[*Examples. Every inf-sup type is explicit. A mid-rad type is typically implicit.*

As an example of the need for a specified hull algorithm, let $\mathbb{T}$ be the mid-rad type (§11.6.2) where m and r use the same floating point format $\mathbb{F}$, say `binary64`*, and let $\boldsymbol{s}$ be the interval $[-1, 1+\epsilon]$ where $1+\epsilon$ is the next $\mathbb{F}$-number above 1. Clearly any minimal interval (m, r) enclosing $\boldsymbol{s}$ has $r = 1+\epsilon$. But m can be any of the many $\mathbb{F}$-numbers in the range 0 to ϵ; each of these gives a minimal enclosure of $\boldsymbol{s}$.*

A possible general algorithm, for a bounded set $\boldsymbol{s}$ and a mid-rad type, is to choose $m \in \mathbb{F}$ as close as possible to the mathematical midpoint of the interval $[\underline{s}, \overline{s}] = [\inf \boldsymbol{s}, \sup \boldsymbol{s}]$ (with some way to resolve ties) and then the smallest $r \in \mathbb{F}'$ such that $r \geq \max(m - \underline{s}, \overline{s} - m)$. The cost of performing this depends on how the set $\boldsymbol{s}$ is represented. If $\boldsymbol{s}$ is a binary64 inf-sup interval, it is simple. If $\boldsymbol{s}$ is defined as the range of some exotic function, it could be expensive.]

For 754-conforming implementations the hull operations of the inf-sup types derived from the formats `binary32`, `binary64`, `binary128`, `decimal64` and `decimal128` are denoted respectively as

$$\mathrm{hull}_{\mathtt{b32}},\ \mathrm{hull}_{\mathtt{b64}},\ \mathrm{hull}_{\mathtt{b128}},\ \mathrm{hull}_{\mathtt{d64}},\ \mathrm{hull}_{\mathtt{d128}}.$$

11.8.2. *Hull in several dimensions.* In n dimensions the $\mathbb{T}$-hull, as defined mathematically in §11.8.1, is extended to act componentwise, namely for an arbitrary subset $\boldsymbol{s}$ of $\mathbb{R}^n$ it is $\mathrm{hull}_{\mathbb{T}}(\boldsymbol{s}) = (\boldsymbol{y}_1, \ldots, \boldsymbol{y}_n)$ where

$$\boldsymbol{y}_i = \mathrm{hull}_{\mathbb{T}}(\boldsymbol{s}_i),$$

and $\boldsymbol{s}_i = \{\, s_i \mid s \in \boldsymbol{s} \,\}$ is the projection of $\boldsymbol{s}$ on the ith coordinate dimension. It is easily seen that this is a minimal $\mathbb{T}$-box containing $\boldsymbol{s}$, and that if $\mathbb{T}$ is explicit it equals the unique tightest $\mathbb{T}$-box containing $\boldsymbol{s}$.

11.8.3. *Interval type conversion.* For each supported bare interval type $\mathbb{T}$, an implementation shall provide the hull operation $\mathrm{hull}_{\mathbb{T}}(\boldsymbol{x})$ for input intervals $\boldsymbol{x}$ of any supported bare interval type. Conversion of an interval from one type to another, whether explicit or implicit, shall be done by applying the hull operation of the output type.

11.9. Level 2 interval extensions. Let $\mathbb{T}$ be a bare interval type and f an n-variable scalar point function. A **$\mathbb{T}$-interval extension** of f, also called a **$\mathbb{T}$-version** of f, is a mapping $\boldsymbol{f}$ from n-dimensional $\mathbb{T}$-boxes to $\mathbb{T}$-intervals, that is $\boldsymbol{f} : \mathbb{T}^n \to \mathbb{T}$, such that $f(x) \in \boldsymbol{f}(\boldsymbol{x})$ whenever $x \in \boldsymbol{x}$ and $f(x)$ is defined. Equivalently

$$\boldsymbol{f}(\boldsymbol{x}) \supseteq \mathrm{Rge}(f \,|\, \boldsymbol{x}). \tag{34}$$

for any $\mathbb{T}$-box $\boldsymbol{x} \in \mathbb{T}^n$, regarding $\boldsymbol{x}$ as a subset of $\mathbb{R}^n$. Generically, such mappings are called Level 2 interval extensions.

Though only defined over a finite set of boxes, a Level 2 extension of f is equivalent to a full Level 1 extension of f (§9.4.3) so that this document does not distinguish between Level 2 and Level 1 extensions. Namely define $\boldsymbol{f}^*$ by

$$\boldsymbol{f}^*(\boldsymbol{s}) = \boldsymbol{f}(\mathrm{hull}_{\mathbb{T}}(\boldsymbol{s}))$$

for any subset $\boldsymbol{s}$ of $\mathbb{R}^n$. Then the interval $\boldsymbol{f}^*(\boldsymbol{s})$ contains $\mathrm{Rge}(f \,|\, \boldsymbol{s})$ for any $\boldsymbol{s}$, making $\boldsymbol{f}^*$ a Level 1 extension, and $\boldsymbol{f}^*(\boldsymbol{s})$ equals $\boldsymbol{f}(\boldsymbol{s})$ whenever $\boldsymbol{s}$ is a $\mathbb{T}$-box.

In the context of a 754-conforming implementation, *typeOf* operations occur, where the inputs may be a mix of related types (e.g., the inf-sup types of `binary32` and `binary64`). In such cases, $\mathbb{T}$ is deemed to be the union of the individual types involved.

11.10. Accuracy requirements.

In this subclause, *operation* denotes any Level 2 version, provided by the implementation, of a Level 1 operation with interval output and at least one interval input.

The standard makes recommendations, but no requirements, on the accuracy of operations. It makes requirements on documenting the accuracy achieved by an implementation.

11.10.1. *Measures of accuracy.* Three **accuracy modes** are defined: *tightest*, *accurate* and *valid*, that indicate the quality of interval enclosure achieved by an operation. The *tightest* and *valid* modes apply to all interval types and all operations.

Each mode is in the first instance a property of an individual evaluation, that is of a pair $(\boldsymbol{f}, \boldsymbol{x})$ where $\boldsymbol{f}$ is an operation and $\boldsymbol{x}$ is an input box. The *accurate* mode is defined only for inf-sup types (because it involves the `nextOut` function), and for interval forward and reverse arithmetic operations of §9.6.3, 9.6.5 (because *accurate* requires operations $\boldsymbol{f}$ that are monotone at Level 1: $\boldsymbol{u} \subseteq \boldsymbol{v}$ implies $\boldsymbol{f}(\boldsymbol{u}) \subseteq \boldsymbol{f}(\boldsymbol{v})$).

Let $\boldsymbol{f}_{\mathrm{exact}}$ denote the corresponding Level 1 operation and, for a forward or reverse arithmetic operation, let f be the underlying point function[2].

The weakest mode, *valid*, is just the property of enclosure:

$$\boldsymbol{f}_{\mathrm{exact}}(\boldsymbol{x}) \subseteq \boldsymbol{f}(\boldsymbol{x}); \tag{35}$$

this is the minimal requirement for conformance. For a forward arithmetic operation, it is equivalent to (34).

[2]Non-interval arguments, such as the p of $\mathtt{pown}(\boldsymbol{x}, p)$, are ignored in the following

The strongest mode, *tightest*, is the property that $\boldsymbol{f}(\boldsymbol{x})$ equals the value $\boldsymbol{f}_{\text{tightest}}(\boldsymbol{x})$ that gives best possible $\mathbb{T}$-interval enclosure of the Level 1 result:

$$\boldsymbol{f}_{\text{tightest}}(\boldsymbol{x}) = \text{hull}_{\mathbb{T}}(\boldsymbol{f}_{\text{exact}}(\boldsymbol{x})). \tag{36}$$

For a forward arithmetic operation, it is equivalent to

$$\boldsymbol{f}(\boldsymbol{x}) = \text{hull}_{\mathbb{T}}(\text{Rge}(f \mid \boldsymbol{x})). \tag{37}$$

The intermediate mode *accurate* asserts that $\boldsymbol{f}(\boldsymbol{x})$ is *valid*, (35), and is at most slightly wider than the result of applying the *tightest* version to a slightly widened input box:

$$\boldsymbol{f}(\boldsymbol{x}) \subseteq \texttt{nextOut}(\boldsymbol{f}_{\text{tightest}}(\texttt{nextOut}(\text{hull}_{\mathbb{T}}(\boldsymbol{x})))), \tag{38}$$

—the inner $\text{hull}_{\mathbb{T}}$ is explained below.

The **nextOut** function is defined as follows, for an inf-sup type $\mathbb{T}$ derived from a number format $\mathbb{F}$. For any $\mathbb{F}$-number x, define $\texttt{nextUp}(x)$ to be $+\infty$ if $x = +\infty$, and the least member of $\mathbb{F}$ greater than x otherwise; since $\pm\infty$ belong to $\mathbb{F}$ by definition, this is always well-defined. Similarly $\texttt{nextDown}(x)$ is $-\infty$ if $x = -\infty$, and the greatest member of $\mathbb{F}$ less than x otherwise. [*Note. For an IEEE 754 format, these are the* `nextUp` *and* `nextDown` *functions in 754-2008 §5.3.1.*]

Then for a nonempty $\mathbb{T}$-interval $\boldsymbol{x} = [\underline{x}, \overline{x}]$, define

$$\texttt{nextOut}(\boldsymbol{x}) = [\texttt{nextDown}(\underline{x}), \texttt{nextUp}(\overline{x})). \tag{39}$$

For a $\mathbb{T}$-box, **nextOut** acts componentwise.
[*Notes. In (38):*

- *The inner* `nextOut()` *aims to handle the problem of a function like* $\sin x$ *evaluated at a very large argument, where a small relative change in the input can produce a large relative change in the result.*
- *The outer* `nextOut()` *relaxes the requirement for correct (rather than, say, faithful) rounding, which may be hard to achieve for some special functions at some arguments.*
- *The inputs to* $\boldsymbol{f}$ *might be of a different type from the output type* $\mathbb{T}$*. The inner* $\text{hull}_{\mathbb{T}}$ *ensures* $\boldsymbol{x}$ *is widened by "at least an* $\mathbb{F}$*-ulp" (ulp = unit in the last place) at each finite endpoint even when this is so. For example, let* $\mathbb{T}$ *be a 2-digit decimal inf-sup type. Then* `nextOut` *widens the* $\mathbb{T}$*-interval* $\boldsymbol{x} = [2.4, 3.7]$ *to* $[2.3, 3.8]$*—an ulp at each end. But an operation might accept 4-digit decimal inf-sup inputs, and* $\boldsymbol{x}$ *might be* $[2.401, 3.699]$*. Then* $\texttt{nextOut}(\boldsymbol{x})$ *is* $[\texttt{nextDown}(2.401), \texttt{nextUp}(3.699)] = [2.4, 3.7]$*, a widening by a small fraction of an ulp. But* $\texttt{nextOut}(\text{hull}_{\mathbb{T}}([2.401, 3.699])) = \texttt{nextOut}([2.4, 3.7]) = [2.3, 3.8]$*, giving a significant widening.*

]

11.10.2. *Documentation requirements.* An implementation shall document an achieved accuracy for each of its interval operations. This shall be done by dividing the operation's domain of definition into disjoint ranges and stating an accuracy in each range. This information may be supplemented by further detail, e.g., to give accuracy data in a more appropriate way for a non-inf-sup type.
[*Example. For* $\sin(\boldsymbol{x})$ *in the* `binary64` *inf-sup type the accuracy might be given as*

Operation	*Type*	*Mode*	*Range*
sin	`binary64`	*tightest*	*for any* $\boldsymbol{x} \subseteq [-10^{15}, 10^{15}]$
		accurate	*for all other* $\boldsymbol{x}$.

Such information by its nature is conservative; the actual accuracy might be better than stated, for many inputs.]

Each operation should be identified by a language- or implementation-defined name of the Level 1 operation (which may differ from that used in this standard), its output type, its input type(s) if necessary, and any other information (e.g., a library version) needed to resolve ambiguity.

11.10.3. *Recommended accuracies.* For 754-conforming types, the accuracy of operations for arbitrary interval inputs should be at least that given in Table 8, which represents what can be achieved with acceptable efficiency at the time of writing.

(a) Forward

Name	Accuracy
`add`(x, y)	tightest
`sub`(x, y)	tightest
`mul`(x, y)	tightest
`div`(x, y)	tightest
`recip`(x)	tightest
`sqrt`(x)	tightest
`hypot`(x, y)	accurate
`case`(b, g, h)	tightest
`sqr`(x)	tightest
`pown`(x, p)	accurate
`pow`(x, y)	accurate
`exp,exp2,exp10`(x)	accurate
`log,log2,log10`(x)	accurate
`sin`(x)	accurate
`cos`(x)	accurate
`tan`(x)	accurate
`asin`(x)	accurate
`acos`(x)	accurate
`atan`(x)	accurate
`atan2`(y, x)	accurate
`sinh`(x)	accurate
`cosh`(x)	accurate
`tanh`(x)	accurate
`asinh`(x)	accurate
`acosh`(x)	accurate
`atanh`(x)	accurate
`sign`(x)	tightest
`ceil`(x)	tightest
`floor`(x)	tightest
`round`(x)	tightest
`trunc`(x)	tightest
`abs`(x)	tightest
`min`$(x_1, \dots, x_k)$	tightest
`max`$(x_1, \dots, x_k)$	tightest

(b) Reverse

Name	Accuracy
`sqrRev`(c, x)	accurate
`recipRev`(c, x)	accurate
`absRev`(c, x)	accurate
`pownRev`$c, (x, p)$	accurate
`sinRev`(c, x)	accurate
`cosRev`(c, x)	accurate
`tanRev`(c, x)	accurate
`coshRev`(c, x)	accurate
`mulRev`(b, c, x)	accurate
`divRev1`(b, c, x)	accurate
`divRev2`(a, c, x)	accurate
`powRev1`(b, c, x)	accurate
`powRev2`(a, c, x)	accurate
`atan2Rev1`(b, c, x)	accurate
`atan2Rev2`(a, c, x)	accurate

Table 8. Accuracy levels for required arithmetic operations.

11.11. Required operations on bare intervals.

An implementation shall provide a $\mathbb{T}$-version, see §11.9, of each operation listed in subclauses §11.11.1 and §11.11.3 to §11.11.10, for each supported type $\mathbb{T}$. That is, those of its inputs and outputs that are intervals, are of type $\mathbb{T}$.

A 754-conforming implementation, or part thereof, shall provide mixed-type *typeOf* operations, as specified in §11.3.1, for the following operations, which correspond to those that 754 requires to be provided as *formatOf* operations.

`add, sub, mul, div, recip, sqrt, sqr, sign, ceil, floor, round, trunc, abs, min, max, fma.`

An implementation may provide more than one version of some operations for a given type. For instance it may provide an "accurate" version in addition to a required "tightest" one, to offer a trade-off of accuracy versus speed or code size. How such a facility is provided, is language- or implementation-defined.

11.11.1. *Interval literals.*

An *interval literal* is a text string that denotes an interval, termed its value. A *number literal* (within this subclause) means a string that may be part of an interval literal and denotes an extended-real number, termed its value.

Within an interval literal, conversion of a number literal to its value is done as if in infinite precision, ignoring any default or explicit language-defined precision specification within it. Conversion of an interval literal's value $\boldsymbol{x}$ to a finite-precision interval $\boldsymbol{y}$ is a separate operation. $\boldsymbol{y}$ shall in all cases contain $\boldsymbol{x}$; typically $\boldsymbol{y}$ is the $\mathbb{T}$-hull of $\boldsymbol{x}$ for some interval type $\mathbb{T}$.

The following forms of number literal shall be supported:

(1) A number literal[3] of the form provided by the host language of the implementation, in decimal or any other supported radix.
(2) Either of the strings `inf` or `infinity`, ignoring case, optionally preceded by `+`, with value $+\infty$; or preceded by `-`, with value $-\infty$.
(3) A string in the hexadecimal-significand form of IEEE 754-2008, §5.12.3, with the value specified there.
(4) A string p `/` q, that is p and q separated by the `/` character, where p, q are decimal integer strings, with p optionally signed and $q > 0$. Its value is the exact rational number p/q.

[*Notes. These categories need not be mutually exclusive, e.g., in C/C++ (i.e., with that host language), form (3) is included in (1).*]
[*Examples. In C or Java, the number literals* `0.2` *and* `0.2f` *are of form (1) and have the same value, the real number* $\frac{1}{5}$*, despite having default precision "double", and explicit precision "float", respectively. Similarly in Fortran,* `0.2` *and* `2.d-1` *both have value* $\frac{1}{5}$*. For any host language the literals* `0xffp-2` *(form (3)) and* `255/256` *(form (4)) both have the value* $\frac{255}{256}$*.*]

The forms of interval literal that shall be supported are shown in the following list. To simplify stating the needed constraints, e.g. $l \le u$, the number literals l, u, m, r are identified with their values. Space shown between elements of a literal, below, denotes zero or more characters that count as whitespace in the host language. For instance in the mid-rad form, whitespace may occur on either side of m and r but not between the two characters of `+-`. [*Note. Some characters, e.g., newline, may be whitespace in some contexts but not others.*]

Constants: The string `empty`, ignoring case, whose value is the empty set $\emptyset$; and the string `entire`, ignoring case, whose value is the whole line $\mathbb{R}$.
Inf-sup form: a string `[` l `,` u `]` where l and u are number literals with $l \le u$, $l < +\infty$ and $u > -\infty$, see §9.2. Its value is the mathematical interval $[l, u]$.
Mid-rad form: a string `<` m `+-` r `>` where m and r are number literals representing finite numbers with $r \ge 0$. Its value is the mathematical interval $[m - r, m + r]$.
Uncertain form: a string m?rue where: m is a decimal number literal of form (1) above, without exponent; r is empty or is a non-negative decimal integer *ulp-count*; u is empty or is a *direction character*, either `u` (up) or `d` (down); and e is empty or is a decimal *exponent field* for the number m. No whitespace is permitted within the string.

One `ulp` equals one unit in the last place of the number m as written. The literal m? by itself denotes m with a symmetrical uncertainty of $\frac{1}{2}$`ulp`, that is the interval $[m - \frac{1}{2}\texttt{ulp}, m + \frac{1}{2}\texttt{ulp}]$. The literal m?r denotes m with a symmetrical uncertainty of r ulps, that is $[m - r \times \texttt{ulp}, m + r \times \texttt{ulp}]$. Adding `d` or `u` converts this to uncertainty in one direction only, e.g. m?`d` denotes $[m - \frac{1}{2}\texttt{ulp}, m]$ and m?r`u` denotes $[m, m + r * \texttt{ulp}]$. Finally the exponent field if present multiplies the whole interval by the appropriate power of 10, e.g. m?rue denotes $10^e \times [m, m + r \times \texttt{ulp}]$.

[3] "floating constant" in C, "real literal constant" in Fortran, "real numeric literal" in Ada, etc.

[*Examples. Assuming the common types of decimal number literals are provided:*

Form	*Literal*	*Value*
Inf-sup	`[1.e3, 1.1e3]`	$[1000, 1100]$
	`[-Inf, 2/3]`	$[-\infty, 2/3]$
Mid-rad	`<3.56 +- 0.002>`	$[3.558, 3.562]$
	`<5/3 +- 1/15>`	$[8/5, 26/15]$
Uncertain	`3.56?`	$[3.555, 3.565]$
	`3.56?2u`	$[3.56, 3.58]$
	`3.560?2`	$[3.558, 3.562]$
	`3.56?2ue+1`	$[35.6, 35.8]$

]

A formal description of interval literals is by the following grammar (using the notation of 754§5.12.3) which defines an intervalLiteral, subject to the constraints on l, u, m, r stated above. anycase(s) matches all upper/lower case variants of the string s, e.g. anycase("ab") matches any of "ab", "Ab", "aB", "AB".

sign	[+-]
digit	[0123456789]
natural	{digit} +
langNumLit	{number literal of host language}
langNumLitMant	{mantissa of number literal of host language}
langNumLitExp	{exponent of number literal of host language}
sp	{whitespace character of host language} *
infLit	{sign} ? (anycase("inf") | anycase("infinity"))
hexNumLit	{see 754§5.12.3}
ratNumLit	{sign} ? {digit} + "/" {digit} +
numberLiteral	({langNumLit} | {infLit} | {hexNumLit} | {ratNumLit} |)
pointIntvl	"[" {sp} {numberLiteral} {sp} "]"
infSupIntvl	"[" {sp} {numberLiteral} {sp} "," {sp} {numberLiteral} {sp} "]"
midRadIntvl	"<" {sp} {numberLiteral} {sp} "+-" {sp} {numberLiteral} {sp} ">"
uncertIntvl	{langNumLitMant} "?" {natural} ? [du] ? {langNumLitExp} ?
constIntvl	(anycase("empty") | anycase("entire"))
intervalLiteral	({pointIntvl} | {infSupIntvl} | {midRadIntvl} | {uncertIntvl} | {constIntvl})

⚠ Possible enhancements:

– Floating literals may include denotations of real constants such as Pi denoting π.

11.11.2. *Interval constants.* An implementation shall provide a $\mathbb{T}$-version of each constant function in §9.6.2, for each supported bare interval type $\mathbb{T}$. It returns a $\mathbb{T}$-interval.

11.11.3. *Forward-mode elementary functions.* An implementation shall provide a $\mathbb{T}$-version of each forward arithmetic operation in §9.6.3, for each supported bare interval type $\mathbb{T}$. Its inputs and output are $\mathbb{T}$-intervals.

For a 754-conforming type, each such operation shall have an extension of that type with accuracy mode as in Table 8(a). For other types, the accuracy mode is language- or implementation-defined.

[*Note. For operations with some integer arguments, such as integer power x^n, only the real arguments are replaced by intervals.*]

11.11.4. *Interval case expressions and case function.* An implementation shall provide the interval case($\boldsymbol{c}, \boldsymbol{g}, \boldsymbol{h}$) function, see §9.6.4, for each supported type $\mathbb{T}$. The input $\boldsymbol{c}$ is of an arbitrary supported interval type. The inputs $\boldsymbol{g}, \boldsymbol{h}$, and the result, are of type $\mathbb{T}$. The implementation shall be as if the $\mathbb{T}$-version of `convexHull` is used in (10) of §9.6.4.

11.11.5. *Reverse-mode elementary functions.* An implementation shall provide a $\mathbb{T}$-version of each reverse arithmetic operation in §9.6.5, for each supported bare interval type $\mathbb{T}$. Its inputs and output are $\mathbb{T}$-intervals.

For a 754-conforming type, each such operation shall have a version of that type with accuracy mode as in Table 8(b). For other types, the accuracy mode is language- or implementation-defined.

11.11.6. *Cancellative addition and subtraction.* An implementation shall provide a $\mathbb{T}$-version of each of the operations `cancelPlus` and `cancelMinus` in §9.6.6, for each supported bare interval type $\mathbb{T}$. Its inputs and output are $\mathbb{T}$-intervals.

It shall return an enclosure of the Level 1 value if the latter is defined, and Entire otherwise. In particular it shall return Empty if the Level 1 value is Empty. Thus, for the case of $\texttt{cancelMinus}(\boldsymbol{x},\boldsymbol{y})$, it returns Entire in these cases:

– both $\boldsymbol{x}$ and $\boldsymbol{y}$ are unbounded;
– $\boldsymbol{x} \neq \emptyset$ and $\boldsymbol{y} = \emptyset$;
– $\boldsymbol{x}$ and $\boldsymbol{y}$ are nonempty bounded intervals with $\texttt{width}(\boldsymbol{x}) < \texttt{width}(\boldsymbol{y})$.

It returns an unbounded interval, which may be Entire, if the Level 1 value is defined but there is no bounded $\mathbb{T}$-interval containing it.

If $\mathbb{T}$ is a 754-conforming type, the result shall be the $\mathbb{T}$-hull of the Level 1 result when this is defined. [*Note. This may require computing in extra precision in boundary cases: see example in* §*14.6.*]

At user level, implementations should only provide the operations of this subclause in decorated form, to make "no value at Level 1" detectable.

11.11.7. *Set operations.* An implementation shall provide a $\mathbb{T}$-version of each of the operations `intersection` and `convexHull` in §9.6.7, for each supported bare interval type $\mathbb{T}$. Its inputs and output are $\mathbb{T}$-intervals.

These operations shall return the $\mathbb{T}$-interval hull of the exact result.

[*Note. In particular, if $\mathbb{T}$ is an inf-sup type, each operation always returns the exact result. However, this need not be the case with the mixed-type version of the operation, when $\mathbb{T}$ is a 754-conforming type.*]

11.11.8. *Constructors.* There shall be a bare interval version of each constructor in §9.6.8, for each supported bare interval type $\mathbb{T}$. It returns a $\mathbb{T}$-interval.

Both `nums2interval` and the inf-sup form of `text2interval` involve testing if $b = (l \leq u)$ is 0 (false) or 1 (true), to determine whether the interval is empty or nonempty. In the former case, l and u are values of supported number formats within a program; in the latter, they are floating literals.

Evaluating b as 0 when the true value is 1 (a "false negative") leads to falsely returning Empty as an enclosure of the true nonempty interval. Evaluating b as 1 when the true value is 0 (a "false positive") is undesirable, but permissible since it returns a nonempty interval as an enclosure for Empty. Implementations shall ensure that false negatives cannot occur, and should ensure that false positives cannot occur.

A bare interval constructor call either **succeeds** or **fails**. This notion is used to determine the value returned by the corresponding decorated interval constructor. A language- or implementation-defined exception should be signaled when a bare interval constructor call fails.

[*Note. Evaluating b correctly can be hard, if l and u have values very close in a relative sense, and are represented in different ways—e.g., if an implementation allows them to be floating point variables or literals of different radices. It could be especially challenging in an extendable-precision context.*

Language rules can cause such errors even when l and u have the same format. E.g., in C, if `long double` *is supported and has more precision than* `double`*, default behavior might be to round* `long double` *inputs l and u to* `double` *at entry to a* `nums2interval` *call. This is forbidden—the comparison $l \leq u$ requires the exact values to be used, which requires use of a version of* `nums2interval` *with* `long double` *arguments.*]

DRAFT 7.1

nums2interval.

– The inputs l and u to the constructor $\boldsymbol{x} = \texttt{nums2interval}(l,u)$ are datums of supported number formats $\mathbb{F}_l$ and $\mathbb{F}_u$. Mixed format, where $\mathbb{F}_l \neq \mathbb{F}_u$, is possible. For a given $\mathbb{T}$, each $\mathbb{F}_l, \mathbb{F}_u$ combination is called a kind in this subclause.

For all kinds, the result $\boldsymbol{x}$ shall enclose the Level 1 value if this exists, that is, if neither l nor u is NaN, and the exact extended-real values of l and u satisfy $l \leq u$, $l < +\infty$, $u > -\infty$.

The constructor call succeeds if the implementation determines that the Level 1 value exists, or is unable to determine that it does not exist—see the discussion at the start of this subclause. In the latter case the result shall be an interval containing l and u.

The call also succeeds in the special cases $l = u = -\infty$ or $l = u = +\infty$, see below.

Otherwise the call fails, and returns $\boldsymbol{x} = \emptyset$.

- An implementation shall provide at least one kind where l and u have the same format. This format should be compatible with $\mathbb{T}$, see §11.5.
- For $\mathbb{T}$ belonging to a 754-conforming implementation, *formatOf* kinds shall be provided, which accept l and u having any 754 format of that implementation, of the same radix as $\mathbb{T}$'s parent format. The result shall be the $\mathbb{T}$-hull of the Level 1 result, when this exists, and shall be Empty otherwise.
- If $l = u = +\infty$, the Level 1 result is undefined. In finite precision however, l and u are likely to be finite values that overflowed. Therefore in this case `nums2interval` shall return the tightest $\mathbb{T}$-interval that is unbounded above. For any type, this exists and has the form $\boldsymbol{x} = [\texttt{HUGEPOS}, +\infty]$, where `HUGEPOS` is a uniquely defined extended-real number $< +\infty$.

  [*Note. If $\mathbb{T}$ is an inf-sup type based on a format $\mathbb{F}$, then* `HUGEPOS` *is the largest finite $\mathbb{F}$-number. For other types,* `HUGEPOS` *can be $-\infty$, hence $\boldsymbol{x} =$* Entire*, if no $\mathbb{T}$-intervals are unbounded above, except* Entire.]

  If $l = u = -\infty$, `nums2interval` shall return $[-\infty, \texttt{HUGENEG}]$, defined similarly.

text2interval.

Input $\boldsymbol{s}$ to the constructor `text2interval(`$\boldsymbol{s}$`)` is a text string. The constructor call succeeds if the implementation determines that $\boldsymbol{s}$ is a valid interval literal with value $\boldsymbol{x}$, see §11.11.1, and returns a $\mathbb{T}$-interval containing $\boldsymbol{x}$. It also succeeds if the implementation evaluates finite bounds l, u but cannot determine that $l \le u$. In the latter case the result shall be an interval containing l and u. Otherwise the call fails and the result is Empty.

If $\mathbb{T}$ is a 754-conforming type, the result shall be the $\mathbb{T}$-hull of $\boldsymbol{x}$.

11.11.9. *Numeric functions of intervals.* An implementation shall provide a $\mathbb{T}$-version of each numeric function in Table 3 of §9.6.9, for each supported bare interval type $\mathbb{T}$. It shall return a result in a supported number format $\mathbb{F}$ as defined in §11.5. Several, user-selectable, versions may be provided, returning results in different formats.

The implementation shall document how it breaks ties, e.g., when computing the closest $\mathbb{F}$-number to a value that is midway between two $\mathbb{F}$-numbers. If $\mathbb{F}$ is a 754-conforming format, the tie-breaking method shall follow 754§4.3.1 and 754§4.3.3; otherwise it is language- or implementation-defined.

If $\mathbb{T}$ is a 754-conforming type, versions shall be provided that return the result in any supported 754 format of the same radix as $\mathbb{T}$.

- `inf(`$\boldsymbol{x}$`)` returns the Level 1 value, rounded toward $-\infty$.
- `sup(`$\boldsymbol{x}$`)` returns the Level 1 value, rounded toward $+\infty$.
- `mid(`$\boldsymbol{x}$`)` returns NaN if $\boldsymbol{x}$ is Empty, and 0 if $\boldsymbol{x}$ is Entire. Otherwise for nonempty $\boldsymbol{x} = [\underline{x}, \overline{x}]$ its value is defined by the following cases.

$\underline{x}, \overline{x}$ both finite	the closest finite $\mathbb{F}$-number to the Level 1 value, breaking ties appropriately;
$\underline{x} = -\infty$, $\overline{x}$ finite	the finite negative $\mathbb{F}$-number of largest magnitude;
$\underline{x}$ finite, $\overline{x} = +\infty$	the finite positive $\mathbb{F}$-number of largest magnitude;

  The three tabulated cases can be obtained by computing $(\underline{x} + \overline{x})/2$ exactly in the extended reals and rounding the result to the nearest finite $\mathbb{F}$-number.
- `rad(`$\boldsymbol{x}$`)` returns NaN if $\boldsymbol{x}$ is empty, and otherwise the smallest $\mathbb{F}$-number r such that $\boldsymbol{x}$ is contained in the exact interval $[m - r, m + r]$, where m is the value returned by `mid(`$\boldsymbol{x}$`)`.

  [*Note.* `rad(`$\boldsymbol{x}$`)` *may be $+\infty$ even though $\boldsymbol{x}$ is bounded, if $\mathbb{F}$ has insufficient range. However, if $\mathbb{F}$ is a 754 format and $\mathbb{T}$ is the derived inf-sup type,* `rad(`$\boldsymbol{x}$`)` *is finite for all bounded nonempty intervals.*]
- `wid(`$\boldsymbol{x}$`)` returns the same value as $2 *$ `rad(`$\boldsymbol{x}$`)`, rounded toward $+\infty$.

  [*Note. At Level 2,* `wid(`$\boldsymbol{x}$`)` *may be infinite though* `rad(`$\boldsymbol{x}$`)` *is finite.*]
- `mag(`$\boldsymbol{x}$`)` returns the Level 1 value rounded toward $+\infty$ if this value exists (if $\boldsymbol{x}$ is nonempty). Otherwise it returns NaN.
- `mig(`$\boldsymbol{x}$`)` returns the Level 1 value rounded toward zero if this value exists (if $\boldsymbol{x}$ is nonempty). Otherwise it returns NaN.

11.11.10. *Boolean functions of intervals.*

An implementation shall provide a $\mathbb{T}$-version of the function `isEmpty(`$\boldsymbol{x}$`)` and the function `isEntire(`$\boldsymbol{x}$`)` in §9.6.10, for each supported bare interval type $\mathbb{T}$.

An implementation shall provide a $\mathbb{T}$-version of each of the comparison relations in Table 4 of §9.6.10, for each supported bare interval type $\mathbb{T}$. Its inputs are $\mathbb{T}$-intervals.

For a 754-conforming part of an implementation, mixed-type versions of these relations shall be provided, where the inputs have arbitrary 754-conforming types of the same radix.

These comparisons shall return, in all cases, the correct value of the comparison applied to the intervals represented by the inputs as if in infinite precision. In particular `equal` shall return `true` if and only if its arguments are equal as defined in §11.4.

11.11.11. *Complete arithmetic, dot product function.* An implementation that provides 754-conforming types shall provide **complete arithmetic**, as specified in U. Kulisch and V. Snyder [**4**], for the parent format $\mathbb{F}$ of at least one such type.

This involves providing a *complete format* datatype $C(\mathbb{F})$ associated with the relevant $\mathbb{F}$, and associated operations. A $C(\mathbb{F})$ datum z holds a fixed-point number of the relevant radix (2 or 10), with enough digits before and after the point to let multiply-add operations $z + x * y$ be done exactly, where x and y are arbitrary finite $\mathbb{F}$-numbers. It also holds one bit for sign, and 3 bits for status information (equivalent to a decoration).

[*Example. For the* `binary64` *format the recommended complete format has 4 bits for sign and status, 2134 bits before the point, and 2150 after the point, for a total of 4288 bits or 536 bytes; this allows for at least* 2^{88} *multiply-adds before overflow can occur.*]

The following operations shall be provided, see [**4**] for details.

– `convert` converts from a complete format to a floating point format, or vice versa, or from one complete format to another.
– `completeAddition` and `completeSubtraction` add or subtract two complete or floating point format operands, of which at least one is complete, giving a complete format result.
– `completeMultiplyAdd` computes $z + x * y$ where z has a complete format and x, y are of floating point format, giving a complete format result.
– `completeDotProduct`. Let a and b be vectors of length n holding floating point numbers of format $\mathbb{F}$. Then `completeDotProduct`(a, b) computes $a \cdot b = \sum_{k=1}^{n} a_k b_k$ exactly and rounds it once to give a result of format $\mathbb{F}$.

  For this final rounding, all 754 rounding modes should be supported. If correct rounding (to nearest) is not provided then faithful rounding (with an error < 1 ulp) shall be provided.

11.12. Recommended operations.

To come shortly.

12. The decoration system at Level 2

12.1. Decorated interval types. Formally, the **decorated interval type $\mathbb{DT}$ derived** from a bare interval type $\mathbb{T}$ comprises the set of triples $(\boldsymbol{x}, t, d)$ where $(\boldsymbol{x}, t)$ is a $\mathbb{T}$-interval tagged with the name t of $\mathbb{T}$ according to §11.4, and $d \in \mathbb{D}$ is a decoration. The members of $\mathbb{DT}$ are **decorated interval datums**.

By convention, as with bare intervals (§11.4), the name t is omitted except when clarity requires. Thus $\mathbb{T}$ is regarded as a finite set of mathematical intervals, and a member of $\mathbb{DT}$ is written as a pair $(\boldsymbol{x}, d)$, equivalently $\boldsymbol{x}_d$, where $\boldsymbol{x} \in \mathbb{T}$ and $d \in \mathbb{D}$, that follows the rule for permissible combinations in §10.4.

12.2. Required decorated types. An implementation shall provide the derived decorated interval type for each provided bare interval type. Conversely each provided decorated interval type shall be derived from a provided bare interval type. [*Note. That is, the map "is derived from" is a bijection from the set of provided bare interval types to the set of provided decorated interval types, with inverse "is parent of".*]

There shall be a Not an Interval, NaI, datum of each provided decorated interval type. It shall appear to be unique as far as Level 2 operations are concerned, but operations may be provided to set and get a payload in an NaI for diagnostic purposes, in an implementation-defined way (see §10.3).

12.3. Decorated versions of an operation. Let φ be a Level 2 operation whose (input or output) arguments are either of bare interval type, or of a non-interval type (e.g., integer or boolean). A **decorated version** of φ has the same number and order of arguments, with each

argument of bare interval type replaced by an argument of the derived decorated interval type; non-interval argument types are unchanged. Further,
[*Example. The integer power function* pown$(x, p) = x^p$ *has Level 2 bare interval versions with signature* $\mathbb{T}$ = pown$(\mathbb{T}, \mathbb{F})$ *where* $\mathbb{T}$ *is a bare interval type and* $\mathbb{F}$ *an integer format. Its decorated version has signature* $\mathbb{DT}$ = pown$(\mathbb{DT}, \mathbb{F})$ *where* $\mathbb{DT}$ *is the decorated type derived from* $\mathbb{T}$.]

12.4. Required operations on decorated intervals. ⚠ Elsewhere! The function newDec() shall be provided. Its input is a $\mathbb{T}$-interval or $\mathbb{DT}$-interval and its output is a $\mathbb{DT}$-interval as specified in §10.5.

12.4.1. *Interval literals.* A decorated interval literal string $\boldsymbol{s}$ is defined to consist of an interval-string sx followed by an underscore followed by a decoration string sd. If sx is a valid interval literal (§11.11.1) representing an interval $\boldsymbol{x}$ according to the implementation, and sd is one of ill, emp, trv, def, dac or com representing the corresponding decoration dx, and if $\boldsymbol{x}_{dx}$ is a permitted combination according to §10.4, then $\boldsymbol{s}$ is **valid** and its value is $\boldsymbol{x}_{dx}$. In all other cases $\boldsymbol{s}$ is **invalid** and its value is NaI. If the implementation does not support com, then com is treated as if it were dac.
[*Note. An implementation may implicitly promote a bare interval literal with value* $\boldsymbol{x}$ *to the decorated interval* newDec$(\boldsymbol{x})$ *in suitable contexts, in a language-defined way.*]

12.4.2. *Interval constants.* Each provided Level 2 interval constant (§11.11.2), returning the $\mathbb{T}$-interval Empty or Entire, shall have a decorated version that returns the $\mathbb{DT}$-interval newDec(Empty) or newDec(Entire), respectively.

12.4.3. *Forward-mode elementary functions.* Each provided Level 2 arithmetic operation (§11.11.3) with arguments of type $\mathbb{T}$, shall have a decorated version with corresponding arguments of type $\mathbb{DT}$. It shall be a decorated interval extension as defined in §10.6—thus the interval part of its output is the same as if the bare interval operation were applied to the interval parts of its inputs.

The only freedom of choice in the decorated version is how the local decoration, denoted dv_0 in (25) of §10.6, is computed. dv_0 shall be the strongest possible (and is thus uniquely defined) if the accuracy mode of the corresponding bare interval operation is "tightest", but otherwise is only required to obey (25).

12.4.4. *Interval case expressions and case function.* ⚠ TBW. Arnold Neumaier had a special recipe, which I need to look up.

12.4.5. *Interval-valued non-arithmetic operations.* This set of operations includes the reverse-mode elementary functions of §11.11.5, the cancellative addition and subtraction of §11.11.6 and the set operations of §11.11.7. Each provided Level 2 operation of this set, with arguments of type $\mathbb{T}$, shall have a decorated version with corresponding arguments of type $\mathbb{DT}$. The decoration shall be as in §10.7.

Versions with alternative decorations such as intersectionDec and convexHullDec, if provided, should be provided for all supported types.

12.4.6. *Constructors.* There shall be a decorated version of each provided bare interval constructor of a type $\mathbb{T}$. It returns a $\mathbb{DT}$-interval.

nums2interval.
For any inputs, if the bare interval constructor succeeds as defined in §11.11.8, and returns a $\mathbb{T}$-interval $\boldsymbol{x}$, the decorated version shall return newDec$(\boldsymbol{x})$. Otherwise it shall return NaI.

text2interval.
If the input string $\boldsymbol{s}$ is a valid decorated interval literal as defined in §12.4.1, with value $\boldsymbol{x}_{dx}$, the constructor shall return $\boldsymbol{x}_{dx}$. If $\boldsymbol{s}$ is a valid bare interval literal as defined in §11.11.1, with value $\boldsymbol{x}$, the constructor shall return newDec$(\boldsymbol{x})$. Otherwise it shall return NaI.

12.4.7. *Numeric functions of intervals.* There shall be a decorated version of each provided numeric function of $\mathbb{T}$-intervals, see §11.11.9. It takes $\mathbb{DT}$-interval input and the result format is that of the bare interval operation.

Following §10.7, if no input is NaI, the result is obtained by discarding the decoration and applying the corresponding bare interval operation. In the case of NaI input, the result is NaN if the result format supports this, else is language- or implementation-defined.

12.4.8. *Boolean functions of intervals.*
There shall be a decorated version of each provided boolean function of $\mathbb{T}$-intervals, see §11.11.9. It takes $\mathbb{DT}$-interval input and the result is boolean.

For the functions in the preceding paragraph, following §10.7, if no input is NaI then the result is obtained by discarding the decoration and applying the corresponding bare interval operation. In the case of NaI input, the result is `false`, and a language- or implementation-defined exception should be signaled.

There shall be a function `isNaI`($\boldsymbol{X}$) with $\mathbb{DT}$-interval input $\boldsymbol{X}$, that returns `true` if $\boldsymbol{X}$ is NaI, else `false`. It shall not raise an exception, whether or not its input is NaI.

12.5. Compressed arithmetic at Level 2. To come shortly.

13. Input and output (I/O) of intervals

13.1. Overview. This clause follows the spirit of the scheme for conversion between floating point numbers and character sequences in the 754 standard, where §5.4.2 and §5.12 specify the mathematical properties of conversion while leaving details, mostly of formatting, language- or implementation-defined. It aims to minimise the risk of incompatible implementations of I/O, while allowing languages and compilers some freedom.

The term *text* denotes character sequences generally, in a language-defined character set, and *string* denotes a particular finite character sequence. The standard is concerned with the conversion between intervals internal to a program, and text. It says nothing about how text may be read from or written to a character stream.

For intervals, containment must hold on both input and output so that, when a program computes an enclosure of some quantity given an enclosure of the data, this remains true all the way from text data to text results.

In addition to normal I/O, the standard requires each interval type $\mathbb{T}$ to have a *public representation*. This has two parts: operations to convert any internal $\mathbb{T}$-interval $\boldsymbol{x}$ to a string $\boldsymbol{s}$, and back again to recover $\boldsymbol{x}$ exactly; and documentation of how to convert $\boldsymbol{s}$ to the Level 1 interval represented by $\boldsymbol{x}$. For proprietary types in particular, this makes explicit the mathematical definition of the type, while letting its Level 3 implementation remain private.
[*Note. It follows from the "equal inputs give equal outputs" principle of §11.4 that text output of an interval $\boldsymbol{x}$, whether by* `interval2text` *or by* `interval2public`*, never betrays which of possible alternative internal representations of $\boldsymbol{x}$ was used. E.g., whether a zero bound of an inf-sup interval is stored as -0 or $+0$, or the quantum of a decimal bound (754§2.1.44), shall not be detectable.*]

13.2. Input. Implementations shall provide for each supported interval type $\mathbb{T}$ a function

type-`text2interval`($\boldsymbol{s}$),

where $\boldsymbol{s}$ is a string. If $\boldsymbol{s}$ is a valid interval literal with value $\boldsymbol{x}$, the returned value is a $\mathbb{T}$-interval enclosing $\boldsymbol{x}$. If $\boldsymbol{s}$ is invalid, Empty is returned.

If $\mathbb{T}$ is a 754-conforming type, the returned value shall be the $\mathbb{T}$-hull of $\boldsymbol{x}$. The tightness of enclosure for other types is language- or implementation-defined.

13.3. Output. Implementations shall provide a function

`interval2text`($\boldsymbol{x}$, cs)

where $\boldsymbol{x}$ is a bare interval of any supported type $\mathbb{T}$ and cs is a string, the conversion specifier. It converts $\boldsymbol{x}$ to a valid interval literal $\boldsymbol{s}$ whose value encloses $\boldsymbol{x}$, in a way specified by cs.

The allowed forms of cs are language-defined, and may depend on $\mathbb{T}$, but shall let the user specify any of the following forms for $\boldsymbol{s}$, see §11.11.1.

(i) Inf-sup form `[` l `,` u `]`, where the layouts of l and u can be specified independently.
(ii) Mid-rad form `<` m `+-` r `>`, where the layouts of m and r can be specified independently.
(iii) ⚠ Some of the forms given in the Vienna proposal, Part 6—to be decided.

In either of cases (i) and (ii) the resulting string shall be a valid interval literal.

Here layout of a number means the way it is output as a string. It shall be possible to specify output to a given number of places after the point or to a given number of significant figures. (For instance, by conversion specifiers like `f12.5` and `e12.5` in Fortran, or `%12.5f` and `%12.5e` in C.)

If $\mathbb{T}$ is a 754-conforming type, the enclosure represented by $\boldsymbol{s}$ shall be tightest possible. Namely let $\boldsymbol{x} = [\underline{x}, \overline{x}]$ be a $\mathbb{T}$-interval. For inf-sup form, l is the largest number of the specified layout that is $\leq \underline{x}$ and u is the smallest number of the specified layout that is $\geq \overline{x}$; either may be infinite in case of overflow. For mid-rad form, m is the number of the specified layout that is closest to the exact midpoint; then r is the smallest number of the specified layout such that the exact interval $[m - r, m + r]$ contains $\boldsymbol{x}$. The treatment of infinite values, overflow and tie-breaking shall follow that of the `inf`, `sup`, `mid` and `rad` functions in §11.11.9.

For other types the tightness of enclosure of $\boldsymbol{x}$ by $\boldsymbol{s}$ is language- or implementation-defined.

13.4. Public representation. For any supported interval type $\mathbb{T}$ an implementations shall provide functions *type*-`interval2public` and *type*-`public2interval`, as follows.

– For any $\mathbb{T}$-interval datum $\boldsymbol{x}$ the value *type*-`interval2public`($\boldsymbol{x}$) is a string $\boldsymbol{s}$, the **public representation** of $\boldsymbol{x}$. It is such that *type*-`public2interval`($\boldsymbol{s}$) is the same as $\boldsymbol{x}$ in the sense of §11.4.

  The string $\boldsymbol{s}$ need not contain anything (such as a typename) to identify the type $\mathbb{T}$.

– The implementation's documentation shall explain how to convert $\boldsymbol{s}$ to the mathematical interval $[l, u]$ represented by the datum $\boldsymbol{x}$. This shall comprise an effective algorithm for obtaining l and u as decimal or binary numbers, exactly or to any desired accuracy.

  A public representation should (this is subjective) be simple. For instance if $\boldsymbol{x}$ represents an interval with small integer bounds such as $[1, 2]$, it should be straightforward to convert $\boldsymbol{s}$ by hand or with the help of a pocket calculator. A good public representation exposes the values of the parameters on which the mathematical model of the type is based.

  [*Example. Suppose $\mathbb{T}$ is an inf-sup type whose bounds l, u are rational numbers stored to a fixed number of bits in "floating slash" form. That is, $l = p_l/q_l$ where p_l and q_l ($q_l > 0$) are integers written in binary, occupying k_l and $m - k_l$ bits respectively where k_l (the slash-position) is an integer between 0 and m, and m is a constant of the type. u is similar, involving integers k_u, p_u, q_u. The numbers $k_l, p_l, q_l, k_u, p_u, q_u$ are the parameters of the mathematical model. A good public representation might consist of hexadecimal forms of these six numbers, embedded in suitable punctuation characters.*]

If $\mathbb{T}$ is a 754-conforming type then the public representation $\boldsymbol{s}$ of $\boldsymbol{x}$ shall be a valid interval literal (§11.11.1) that, for nonempty $\boldsymbol{x}$, is of inf-sup form. Its bounds l, u shall be represented as exact decimal numbers if $\mathbb{T}$ is a decimal type, or in the hexadecimal-significand form of 754§5.12.3 if $\mathbb{T}$ is a binary type.

This is "EXTRA BITS" — much will be discarded from final text.

⚠ Q for the "Operations on decorated intervals" clause: We need to specify how the interval part of a `ill` interval behaves w.r.t. real-valued functions like radius (always NaN?) . Also w.r.t. "forget decoration" (gives Empty?).

13.5. Representations.

[*Note. Some of these ideas appeared in previous drafts of the standard or in position papers, others didn't.*]

- A *representation* of an interval type comprises a set $\overline{\mathbb{IF}}$ called an interval level-3 type, and whose members are called (level 3) *interval objects*, together with a map r from a subset of $\overline{\mathbb{IF}}$ (the "valid" objects) to $\mathbb{T}$. If object X maps to datum $\boldsymbol{x}$, we say X represents $\boldsymbol{x}$. (DS§3.1 Table 1 note b: "Not every interval object necessarily represents an interval datum, but when it does, that datum is unique. Each interval datum has at least one representation, and may have more than one.")
  [*Note. A representation is not an approximation—it means just what it says. E.g., an interval object in mid-rad form with midpoint* $m = 1$ *and radius* $r = 1\text{e–}300$ *means precisely the mathematical interval* $[1 - 10^{-300}, 1 + 10^{-300}]$.]
- A *text representation* of an interval type is a representation whose interval level-3 type is a set $\overline{\mathbb{IT}}$ of text strings.
- A representation is *standardized* if there is also provided a map s from the whole of $\mathbb{T}$ into $\overline{\mathbb{IF}}$, such that $s(\boldsymbol{x})$ is a representation of $\boldsymbol{x}$:

$$r(s(\boldsymbol{x})) = \boldsymbol{x} \quad \text{for each } \boldsymbol{x} \text{ in } \mathbb{T}.$$

  The object $s(\boldsymbol{x})$ is called the "standard representation" of the datum $\boldsymbol{x}$.
  [*Example. In an inf-sup representation,* r *might map both the objects* `(-0,3)` *and* `(+0,3)` *to the interval datum* $[0, 3]$. *Suppose the standardized representation always uses* `+0`*; then* s *would map* $[0, 3]$ *to* `(+0,3)`.]
  [*Note. A* standardized text representation *of* $\mathbb{T}$ *essentially defines a way to write any* $\mathbb{T}$*-interval out in text form and read it back* exactly. *Think of map* s *as "write" and* r *as "read". See §13.6.2.*]

13.6. Rationale for defined hulls, text representation, and reproducibility.

13.6.1. *The hull.* The decision whether the hull operation is made part of an interval type's definition affects (a) inter-interval type conversion, which is done by forming the hull; (b) the definition of "tightest" standard functions; (c) hence reproducibility.

[*Example. Use the notation* $m{\pm}r$ *to mean the interval* $[m-r, m+r]$ *written in mid-rad form. Let* $\mathbb{T}$ *comprise all intervals* $m{\pm}r$ *where* m *and* r *belong to the set* $\mathbb{F}$ *of 4-digit decimal floating point numbers, with some finite exponent range that is irrelevant here.*

What is the conversion of the inf-sup interval $[1, 1.003]$ *to mid-rad? If* $\text{hull}_{\mathbb{T}}$ *is not part of the definition of* $\mathbb{T}$*, one implementation can choose* $1.001{\pm}0.002000$*, another can choose* $1.002{\pm}0.002000$*, and both are right.*]

Whether one approves of this non-uniqueness depends on one's philosophy of the standard: should it specify minimum demands consistent with the FTIA, or should it tie things down more closely?

Personally I agree with Dan Zuras (754 chair for a number of years) that one of the main reasons why the 754 floating point standard has been so successful is because it took the hard road of specifying things down to the last bit. The parallel LIA standard, which didn't, has sunk with barely a trace.

So let's tie things down. What, and how far? I think requiring mid-rad implementors to define an unambiguous, platform-independent hull operation is not too much to ask. Nate Hayes agrees (Clause **??** item 7).

13.6.2. *Standardised text representation.* In Motion 17, persuaded by Michel Hack, I included the requirement that—for a 754-conforming inf-sup type $\mathbb{T}$ only—there should be a way to dump any $\mathbb{T}$-interval $\boldsymbol{x}$ to text using 754's "hexadecimal significand" form; but we forgot to require it should be readable back again.

The current motion goes beyond inf-sup to allow *any level 3 representation whatever* of intervals, which, I believe, makes it especially important that there should be an exact textual representation $\boldsymbol{y}$ of any interval datum $\boldsymbol{x}$, documented so that the user can, if desired, manually construct the mathematical interval represented. The two-way mapping (functions r and s) makes it possible to confirm that what one sees as text is what is actually being computed with.

The requirement that $\boldsymbol{y}$ depend only on the level 2 datum $\boldsymbol{x}$, not on the possibly non-unique internal representation of $\boldsymbol{x}$, is deliberate. It simplifies matters for the user and protects the implementer.

13.6.3. *Reproducibility.* I believe reproducibility, important for floating point, is doubly important for interval computing. With default compiler options, running the same interval code on different platforms is not expected to produce the same results down to the last bit. But a user should be able to choose a mode that (presumably at the expense of speed) ensures identical results on all platforms, for code restricting itself to some subset of language features.

There is a counter-argument that reproducibility is *less* important for interval computing: if different platforms give different results, good, because they both enclose the true result. I accept that but am unmoved by it. As Dan Zuras (13 July 2010) says, why should I trust *either* result? For instance, what if a specialised interval computing chip has something like the famous Pentium bug? The bug will probably be far easier to find if I can run the chip in "reproducibility mode".

The key to reproducibility is reproducible behaviour of interval standard functions. The only reasonable way to specify this is to require these functions to return "tightest" results for all arguments. The remarkable work of the French experts means it will soon be practical to compute results correct to the last bit in either rounding direction for all (point) standard functions, all arguments and all sensible number formats, with little loss of speed. For inf-sup interval types this makes "tightest", hence reproducible, standard functions entirely practicable.

For general interval types I do not know how hard it is to achieve the same level of tightness (but see Nate Hayes' view below). In general, I believe any steps that promote reproducibility will in the long term make systems programmers and users alike more confident that our interval systems are correct.

14. Level 3 description

14.1. Representation of intervals by lower/upper bounds.

An implementation may choose any means to represent a level 2 interval datum $\boldsymbol{x}$, provided that it shall be possible to retrieve the bounds of any nonempty $\boldsymbol{x}$ exactly. This is captured by the following definition.

A **concrete interval format (ci-format)** is a surjective mapping from a set C of instances of a data structure to an associated level 2 i-format.

[*Note. Typical choices are*

– **inf-sup** *representation. The data structure is an ordered pair (f1, f2) of floating point datums. A nonempty* $\mathbb{F}$*-interval* $\boldsymbol{x} = [\underline{x}, \overline{x}]$ *is represented by* $(\underline{x}, \overline{x})$.

– **neginf-sup** *representation. As the previous, but* $\boldsymbol{x} = [\underline{x}, \overline{x}]$ *is represented by* $(-\underline{x}, \overline{x})$.

The above two ci-formats use essentially the same data structure, and represent the same i-format, but the mappings are different.

Multi-precision interval packages may represent an interval $\boldsymbol{x} = [\underline{x}, \overline{x}]$ *by a triple* $(\hat{x}, \underline{\delta}, \overline{\delta})$ *where* $\hat{x}$ *is some point in* $\boldsymbol{x}$*, and* $\underline{\delta}$ *and* $\overline{\delta}$ *are very small numbers, and* $\boldsymbol{x} = [\hat{x} + \underline{\delta}, \hat{x} + \overline{\delta}]$ *exactly; or in other ways.*
]

14.2. Format conversion.

On 754 systems, level 2 interval format conversion (the hull operation) shall be implemented in terms of the floating-point operations *formatOf*-`convertFormat` defined in 754§5.4.2, with the appropriate outward rounding.

14.3. Interchange formats.

⚠ We need a motion on this subclause, which was my invention.

The purpose of interchange formats is to allow the loss-free exchange of level 2 interval data between 754-conforming implementations. This is done by imposing a standard level 3 and level 4 representation. Let $\mathbb{F}$ be a 754 format and $\boldsymbol{x}$ a (bare) nonempty $\mathbb{F}$-interval datum, so that its lower bound $\underline{x}$ and upper bound $\overline{x}$ are $\mathbb{F}$-numbers. An interchange format of $\boldsymbol{x}$ is the concatenation of the bit strings of the $\mathbb{F}$-representations of $\underline{x}$ and $\overline{x}$ in that order, where:

- 0 shall be represented as $+0$.
- For decimal formats, any member of the number's cohort is permitted. The choice is implementation-defined.
- When $\boldsymbol{x}$ is the empty set, $\underline{x}$ and $\overline{x}$ are taken as NaN. Whether qNaN or sNaN is used, and any payload, are implementation-defined.

[*Note. The above rules imply an interval has a unique interchange representation if it is nonempty and in a binary format, but not generally otherwise. The reason for the rules is that the sign of a zero endpoint cannot convey any information relevant to intervals; but an implementation may potentially use cohort information, or a* NaN *payload.*]

The interchange format for a decoration comprises ⚠ TBW. Thus a decoration occupies one byte.

The interchange format for a decorated interval is the concatenation of those for its interval and decoration parts, in that order.

A 754-conforming implementation shall provide an interchange format for each supported 754 interval format. Interchange formats for non-754 interval formats, and on non-754 systems, are implementation-defined. If an implementation provides other decoration attributes besides the standard ones, then how it maps them to an interchange format is implementation-defined.

14.4. Support levels for interval elementary functions.

The Fundamental Theorem of Interval Arithmetic (FTIA) relies on each point elementary function e in a real expression being replaced by an interval version $\boldsymbol{e}$. Mathematically, $\boldsymbol{e}$ may be an arbitrary interval extension of e, and its arguments and result are not limited by any concrete interval format.

A level 2 interval version is implemented at level 3 in terms of concrete formats such as binary64. The standard defines three levels of mixed format support, below. For each one, $\boldsymbol{e}$ delivers a result of a specified ci-format $\mathbb{F}$, from operands of a limited number of ci-formats.

Implementations shall give at least SRSF support to all supported elementary functions. 754-conforming implementations shall give at least SRMF support.

SRSF. Single-radix, single-format. In SRSF support, e has an interval version $\boldsymbol{e}$ that takes $\mathbb{F}$-interval operand(s) and gives an $\mathbb{F}$-interval result. Thus explicit format conversion is needed for any operand of a different format from $\mathbb{F}$.
SRMF. Single-radix, mixed-format. In SRMF support, e has an interval version $\boldsymbol{e}$ that takes operand(s) of any supported interval format of the same radix as $\mathbb{F}$, and gives an $\mathbb{F}$-interval result. Thus explicit format conversion is needed for any operand whose format has a different radix from that of $\mathbb{F}$.
MRMF. Mixed-radix, mixed-format. In MRMF support, e has an interval version $\boldsymbol{e}$ that takes operand(s) of any supported interval format, and gives an $\mathbb{F}$-interval result. Thus no explicit format conversion is required for any operand.

SRMF includes SRSF (SRMF support provides SRSF in particular); and MRMF includes SRMF.

MRMF support and mixed-format interval expressions of more than one operation are considered to be language issues.

[*Note. For a 754* formatOf *floating-point operation, for any combination of input and output formats of the same radix, the correctly rounded result is produced, eliminating the risk of "double rounding" error in mixed-format operations.*
Most algorithms for the basic interval operations, in particular those in §14.5, can exploit the formatOf *feature. That is, they can be written in terms of point operations so that arbitrary mixed formats of the same radix can be handled by essentially the same code, while remaining optimally tight at the level of a single interval operation.*]

14.5. Operation tables for basic interval operations.

The tables in this subclause are an explicit realization of the general definition of interval operations given in §9.4.3. They are not normative, but are one possible basis for coding the interval versions of $+$, $-$, $*$, $/$. For fuller details see [1], [2].

Notation. In addition to the notation in §4.1 this subclause also uses, for a specified n-format $\mathbb{F}$:

$\triangledown$, $\triangle$	: the roundings downwards and upwards to the next element of $\mathbb{F}$,
$\overset{\triangledown}{+}$, etc.	: the operations for elements of $\mathbb{F}$ with rounding downwards,
$\overset{\triangle}{+}$, etc.	: the operations for elements of $\mathbb{F}$ with rounding upwards.
$\diamond\ \boldsymbol{s}$	: the same as $\mathrm{hull}_{\mathbb{F}}(\boldsymbol{s})$, the $\mathbb{F}$-hull of a subset $\boldsymbol{s}$ of $\mathbb{R}$.

For intervals $\boldsymbol{a}$ and $\boldsymbol{b} \in \mathbb{IR}$ (the bounded, nonempty mathematical intervals), arithmetic operations are defined as set operations in $\mathbb{R}$ by:

$$\boldsymbol{a} \circ \boldsymbol{b} := \{\, a \circ b \mid a \in \boldsymbol{a} \ \wedge\ b \in \boldsymbol{b} \ \wedge\ a \circ b \text{ is defined}\,\}, \tag{40}$$

for all $\boldsymbol{a}$ and $\boldsymbol{b} \in \mathbb{IR}$ and $\circ \in \{+,-,*,/\}$. If $0 \notin \boldsymbol{b}$ in case of division, then for all $\boldsymbol{a}$ and $\boldsymbol{b} \in \mathbb{IR}$ also $\boldsymbol{a} \circ \boldsymbol{b} \in \mathbb{IR}$.

Then binary arithmetic operations in $\mathbb{IF}$ (the bounded, nonempty level 2 interval datums) are uniquely defined by:

$$a \overset{\diamond}{\circ} b := \ \diamond\ (a \circ b), \tag{41}$$

for all $\boldsymbol{a}$ and $\boldsymbol{b} \in \mathbb{IF}$ and all $\circ \in \{+,-,*,/\}$. For division we assume again that $0 \notin \boldsymbol{b}$.

For intervals $\boldsymbol{a} = [a_1, a_2]$ and $\boldsymbol{b} = [b_1, b_2] \in \mathbb{IF}$ these operations $\overset{\diamond}{\circ}$, for $\circ \in \{+,-,*,/\}$, have the property

$$a \overset{\diamond}{\circ} b = \left[\min_{i,j=1,2}(a_i \overset{\triangledown}{\circ} b_j),\ \max_{i,j=1,2}(a_i \overset{\triangle}{\circ} b_j) \right],$$

or with the monotone roundings $\triangledown$ and $\triangle$,

$$a \overset{\diamond}{\circ} b = \left[\ \triangledown \min_{i,j=1,2}(a_i \circ b_j),\ \triangle \max_{i,j=1,2}(a_i \circ b_j) \right].$$

These operations and the unary operation $-\boldsymbol{a}$ can be expressed by more explicit formulas as shown in Tables 9–12. There the operators for intervals are simply denoted by $+, -, *$, and $/$.

These tables assume that $\boldsymbol{a}$ and $\boldsymbol{b}$ are nonempty and bounded. To extend them to general intervals, the first rule is that any operation with the empty set $\emptyset$ returns the empty set. Then, the tables extend to possibly unbounded intervals of $\overline{\mathbb{IF}}$ by using the standard formulae for arithmetic operations involving $\pm\infty$, which are implemented in 754, together with one rule that goes beyond 754 arithmetic:

$$0 * (-\infty) = (-\infty) * 0 = 0 * (+\infty) = (+\infty) * 0 = 0.$$

This rule is not a new mathematical law, merely a short cut to compute the bounds of the result of multiplication on unbounded intervals.

Negation	$-\boldsymbol{a} = [-a_2, -a_1]$.
Addition	$[a_1, a_2] + [b_1, b_2] = [a_1 \overset{+}{\triangledown} b_1,\ a_2 \overset{+}{\vartriangle} b_2]$.
Subtraction	$[a_1, a_2] - [b_1, b_2] = [a_1 \overset{-}{\triangledown} b_2,\ a_2 \overset{-}{\vartriangle} b_1]$.

Table 9. Negation, addition, subtraction.

Multiplication $[a_1, a_2] * [b_1, b_2]$	$[b_1, b_2]$ $b_2 \leq 0$	$[b_1, b_2]$ $b_1 < 0 < b_2$	$[b_1, b_2]$ $b_1 \geq 0$
$[a_1, a_2], a_2 \leq 0$	$[a_2 \overset{*}{\triangledown} b_2, a_1 \overset{*}{\vartriangle} b_1]$	$[a_1 \overset{*}{\triangledown} b_2, a_1 \overset{*}{\vartriangle} b_1]$	$[a_1 \overset{*}{\triangledown} b_2, a_2 \overset{*}{\vartriangle} b_1]$
$a_1 < 0 < a_2$	$[a_2 \overset{*}{\triangledown} b_1, a_1 \overset{*}{\vartriangle} b_1]$	$[\min(a_1 \overset{*}{\triangledown} b_2, a_2 \overset{*}{\triangledown} b_1),$ $\max(a_1 \overset{*}{\vartriangle} b_1, a_2 \overset{*}{\vartriangle} b_2)]$	$[a_1 \overset{*}{\triangledown} b_2, a_2 \overset{*}{\vartriangle} b_2]$
$[a_1, a_2], a_1 \geq 0$	$[a_2 \overset{*}{\triangledown} b_1, a_1 \overset{*}{\vartriangle} b_2]$	$[a_2 \overset{*}{\triangledown} b_1, a_2 \overset{*}{\vartriangle} b_2]$	$[a_1 \overset{*}{\triangledown} b_1, a_2 \overset{*}{\vartriangle} b_2]$

Table 10. Multiplication.

Division, $0 \notin \boldsymbol{b}$ $[a_1, a_2]/[b_1, b_2]$	$[b_1, b_2]$ $b_2 < 0$	$[b_1, b_2]$ $b_1 > 0$
$[a_1, a_2], a_2 \leq 0$	$[a_2 \overset{/}{\triangledown} b_1, a_1 \overset{/}{\vartriangle} b_2]$	$[a_1 \overset{/}{\triangledown} b_1, a_2 \overset{/}{\vartriangle} b_2]$
$[a_1, a_2], a_1 < 0 < a_2$	$[a_2 \overset{/}{\triangledown} b_2, a_1 \overset{/}{\vartriangle} b_2]$	$[a_1 \overset{/}{\triangledown} b_1, a_2 \overset{/}{\vartriangle} b_1]$
$[a_1, a_2], 0 \leq a_1$	$[a_2 \overset{/}{\triangledown} b_2, a_1 \overset{/}{\vartriangle} b_1]$	$[a_1 \overset{/}{\triangledown} b_2, a_2 \overset{/}{\vartriangle} b_1]$

Table 11. Division by interval not containing 0.

The general rule for computing the set $\boldsymbol{a}/\boldsymbol{b}$ with $0 \in \boldsymbol{b}$ is to remove its zero from the interval $\boldsymbol{b}$ and perform the division with the remaining set. Whenever zero is an endpoint of $\boldsymbol{b}$, the result of the division can be obtained directly from the above table for division with $0 \notin \boldsymbol{b}$ by the limit process $b_1 \to 0$ or $b_2 \to 0$ respectively. The results are shown in the following table. Here, the parentheses stress that the bounds $-\infty$ and $+\infty$ are not elements of the interval.

Division, $0 \in \boldsymbol{b}$ $[a_1, a_2]/[b_1, b_2]$	$\boldsymbol{b} =$ $[0, 0]$	$[b_1, b_2]$ $b_1 < b_2 = 0$	$[b_1, b_2]$ $0 = b_1 < b_2$
$[a_1, a_2] = [0, 0]$	$\emptyset$	$[0, 0]$	$[0, 0]$
$a_1 < 0, a_2 \leq 0$	$\emptyset$	$[a_2 \overset{/}{\triangledown} b_1, +\infty)$	$(-\infty, a_2 \overset{/}{\vartriangle} b_2]$
$[a_1, a_2], a_1 < 0 < a_2$	$\emptyset$	$(-\infty, +\infty)$	$(-\infty, +\infty)$
$0 \leq a_1, 0 < a_2$	$\emptyset$	$(-\infty, a_1 \overset{/}{\vartriangle} b_1]$	$[a_1 \overset{/}{\triangledown} b_2, +\infty)$

Table 12. Division by interval containing 0.

When zero is an interior point of the denominator, the set $[b_1, b_2]$ splits into the distinct sets $[b_1, 0)$ and $(0, b_2]$, and division by $[b_1, b_2]$ actually means two divisions. The results of the two divisions are already shown in Table 11, division with $0 \in \boldsymbol{b}$.

However, in the user's program the two divisions appear as a single operation, as division by an interval $\boldsymbol{b} = [b_1, b_2]$ with $b_1 < 0 < b_2$—an operation that delivers two distinct results.

⚠ Prof Kulisch's motion proposed several ways to handle this situation, but listing them does not seem appropriate for the standard. Suggestions for text here, please.

14.6. Care needed with innerPlus and innerMinus. (informative)
[*Example. Consider inf-sup intervals using 3 decimal digit floating point arithmetic. Let* $\boldsymbol{x} = [.0001, 1]$ *and* $\boldsymbol{y} = [-1, -.0002]$. *Thus* $\boldsymbol{x}$ *is slightly the wider, so* $\boldsymbol{z}_1 = \texttt{innerMinus}(\boldsymbol{x}, \boldsymbol{y})$ *is defined (its exact value is* $[1.0001, 1.0002]$ *whose tightest 3-digit enclosure is* $[1.00, 1.01]$*), while* $\boldsymbol{z}_2 = \texttt{innerMinus}(\boldsymbol{y}, \boldsymbol{x})$ *is not defined. However, one cannot discriminate these cases using naive 3 digit arithmetic. Comparing* $\texttt{width}(\boldsymbol{x})$ *with* $\texttt{width}(\boldsymbol{y})$ *gives the wrong result, because both are computed (rounding upward) as 1.01, suggesting* $\boldsymbol{z}_2$ *is defined. Computing the bounds of* $\boldsymbol{z}_2$*, namely* $[(-1.00 - .0001), (-.0002 - 1.00)]$ *(with outward rounding), also gives the wrong result, namely* $[-1.01, -1.00]$*, again suggesting* $\boldsymbol{z}_2$ *is defined.*
]

Only real or simulated higher precision is guaranteed to give the correct decision in all cases.

14.7. Implementation of bare object arithmetic. (informative)
⚠ To be written.

15. Level 4 description

⚠ Probably does not need to exist in this standard.

CHAPTER 3

Kaucher Intervals

This Chapter contains the standard for the Kaucher interval flavor.
To be included.

ANNEX A

Details of flavor-independent requirements

16. List of required functions

TBW

17. List of recommended functions

TBW

ANNEX B

Including a new flavor in the standard

To be written following discussion with the IEEE MSC working group.

ANNEX C

Reproducibility

To be written.

⚠ Moved from earlier, hoping it is useful here.

[*Example. Suppose the set-based and Kaucher flavors co-operate by sharing a type* $\mathbb{T}$ *whose intervals have lower and upper bounds that are* `binary64` *floating point numbers. Suppose they implement some subset of* $\mathbb{T}$*'s operations in "tightest" mode, returning the smallest* $\mathbb{T}$*-interval that encloses the Level 1 result. This specification makes such operations flavor-independent (when acting on common intervals). Then any common evaluation, that uses only this shared type and this subset of operations, gives identical results in both flavors, modulo the embedding map.*]

ANNEX D

Set-based flavor: decoration details and examples

18. Local decorations of arithmetic operations

18.1. Forward-mode elementary functions. For each of the required functions φ of §9.6, with the decoration scheme `com` > `dac` > `def` > `trv` > `ill` of Clause 10, Tables 1 to 2 give the **strongest local decoration** for arbitrary interval inputs. That is, they give $\operatorname{dec}(\varphi, \boldsymbol{x})$ for an arbitrary input box $\boldsymbol{x}$. The following facts are used to shorten the tables:

– If any input is empty, the decoration is `trv`, so the tables may assume nonempty inputs.
– Functions $\varphi(x_1, x_2, \ldots)$ that are defined and continuous at all real arguments can be handled in a uniform way. This covers the required functions `neg`, `add`, `sub`, `mul`, `fma`, `sqr`, `pown`(x, p) for $p \geq 0$, `exp` and its variants, `sin`, `cos`, `atan`, `sinh`, `cosh`, `tanh`, `asinh` and `abs`, together with `min` and `max` of any number of arguments.

The functions φ in Table 2 have discontinuities at points within their domain of definition. Hence, one must note a distinction between `dac`, which requires that the restriction of φ to the input box $\boldsymbol{x}$ be continuous, and `com`, which makes the stronger requirement that φ be continuous at each point of $\boldsymbol{x}$. [*Example. For* `floor`(x) *on* $[0, \frac{1}{2}]$, `dac` *is true and* `com` *is false.*] For these functions, finding the tightest interval enclosure of the range, and the local decoration, is simplified by noting that all are *increasing step functions*, that is, each one satisfies $\varphi(u) \leq \varphi(v)$ if $u \leq v$, and takes only finitely many values in any bounded interval. Further, each one is defined on the whole real line. For such an φ on an interval $[\underline{x}, \overline{x}]$ it is easy to see that

(a) The restriction of φ to $\boldsymbol{x}$ is continuous iff $\varphi(\underline{x}) = \varphi(\overline{x})$.
(b) φ is continuous at each point of $\boldsymbol{x}$ iff $\varphi(\underline{x}) = \varphi(\overline{x})$ and neither $\underline{x}$ nor $\overline{x}$ is a jump point of φ.

This gives a simple algorithm (given in the Table) for the range and local decoration. It relies only on φ itself and the set J of jump points of φ, so Table 2 merely displays the set J for each function.

18.2. Interval case function.
⚠ JDP March 2013. This is probably wrong now. Subclause 9.6.4 defines the function `case`(c, g, h), and its required bare interval extension. It propagates decorations like other arithmetic operations, with local decoration d where

$$d = \begin{cases} \text{if } \boldsymbol{c} \text{ is empty} & \text{then} \quad \texttt{trv} \\ \text{elseif } \boldsymbol{c} \text{ is a subset of the half- line } x < 0 & \\ \text{or } \boldsymbol{c} \text{ is a subset of the half- line } x \geq 0 & \text{then} \quad \texttt{dac} \\ \text{else} & \qquad\ \ \texttt{def}. \end{cases}$$

[*Note. Comparisons or overlap relations, as a mechanism for handling cases, are incompatible with the decoration concept since there is no way to account for exceptions. The* `case` *function handles decorations correctly. However, in most cases, functions defined using it give very suboptimal enclosures, and it is preferable to use methods illustrated in* §*10.10.*]

19. Examples of use of decorations

This subclause gives a number of examples intended to clarify decoration concepts and algorithms.

1. If $n = 1$, and f is the square root function, then the strongest decoration of $(f, [0, 1])$ is `dac`; of $(f, [-1, 1])$ is `trv`; and of $(f, [-2, -1])$ is `emp`. The expression $f(x) = \sqrt{-1 - x^2}$, as a real function, has no value for any x, so $\operatorname{dec}(f, \boldsymbol{x}) =$ `ill` for all $\boldsymbol{x}$—though evaluation can never find this value, see examples below.

TABLE 1. Local decorations of required forward elementary functions. Normal mathematical notation is used to include or exclude an interval endpoint, e.g., $(-1, 1]$ denotes $\{ x \in \mathbb{R} \mid -1 < x \le 1 \}$. The specification for each function is written as a set of mutually exclusive cases.

Function φ	Strongest local decoration, for all inputs nonempty
Everywhere continuous $\varphi(x_1, x_2, \dots)$	$\begin{cases} \texttt{com} & \text{if inputs bounded, and result bounded at Level 2;} \\ \texttt{dac} & \text{otherwise.} \end{cases}$
$\texttt{div}(x, y)$	$\begin{cases} \texttt{com} & \text{if } 0 \notin \boldsymbol{y}\text{, inputs bounded, and result bounded at Level 2;} \\ \texttt{trv} & \text{if } 0 \in \boldsymbol{y}; \\ \texttt{dac} & \text{otherwise.} \end{cases}$
$\texttt{recip}(x)$	$\begin{cases} \texttt{com} & \text{if } 0 \notin \boldsymbol{x}, \boldsymbol{x} \text{ bounded, and result bounded at Level 2;} \\ \texttt{trv} & \text{if } 0 \in \boldsymbol{x}; \\ \texttt{dac} & \text{otherwise.} \end{cases}$
$\texttt{sqrt}(x)$	$\begin{cases} \texttt{trv} & \text{if } \boldsymbol{x} \not\subseteq [0, +\infty]; \\ \texttt{com} & \text{if } \boldsymbol{x} \subseteq [0, +\infty], \boldsymbol{x} \text{ bounded, and result bounded at Level 2;} \\ \texttt{dac} & \text{otherwise.} \end{cases}$
$\texttt{case}(c, g, h)$	To be done.
$\texttt{pown}(x, p),\ p \ge 0$	"Everywhere continuous" case.
$\texttt{pown}(x, p),\ p < 0$	$\begin{cases} \texttt{com} & \text{if } 0 \notin \boldsymbol{x}, \boldsymbol{x} \text{ bounded, and result bounded at Level 2;} \\ \texttt{trv} & \text{if } 0 \in \boldsymbol{x}; \\ \texttt{dac} & \text{otherwise.} \end{cases}$
$\texttt{pow}(x, y)$	$\begin{cases} \texttt{trv} & \text{if } (\boldsymbol{x}, \boldsymbol{y}) \not\subseteq \mathcal{D}; \\ \texttt{com} & \text{if } (\boldsymbol{x}, \boldsymbol{y}) \subseteq \mathcal{D}\text{, inputs bounded, and result bounded at Level 2;} \\ \texttt{dac} & \text{otherwise;} \end{cases}$ where $\mathcal{D} = \{ (x, y) \mid x > 0, \text{ or } x = 0 \text{ and } y > 0 \}$.
$\texttt{log}, \texttt{log2}, \texttt{log10}(x)$	$\begin{cases} \texttt{trv} & \text{if } \boldsymbol{x} \not\subseteq (0, +\infty]; \\ \texttt{com} & \text{if } \boldsymbol{x} \subseteq (0, +\infty], \boldsymbol{x} \text{ bounded, and result bounded at Level 2;} \\ \texttt{dac} & \text{otherwise.} \end{cases}$
$\texttt{tan}(x)$	$\begin{cases} \texttt{trv} & \text{if } \boldsymbol{x} \not\subseteq \mathcal{D}; \\ \texttt{com} & \text{if } \boldsymbol{x} \subseteq \mathcal{D}, \boldsymbol{x} \text{ bounded, and result bounded at Level 2;} \\ \texttt{dac} & \text{otherwise.} \end{cases}$ where $\mathcal{D} = \mathbb{R} \setminus \{\text{odd multiples of } \pi/2\}$.
$\texttt{asin}(x), \texttt{acos}(x)$	$\begin{cases} \texttt{com} & \text{if } \boldsymbol{x} \subseteq [-1, 1]; \\ \texttt{trv} & \text{otherwise.} \end{cases}$
$\texttt{atan2}(y, x)$	$\begin{cases} \texttt{trv} & \text{if } (\boldsymbol{x}, \boldsymbol{y}) \not\subseteq \mathcal{D}; \\ \texttt{com} & \text{if } (\boldsymbol{x}, \boldsymbol{y}) \subseteq \mathcal{D}\text{, inputs bounded, and result bounded at Level 2;} \\ \texttt{dac} & \text{otherwise;} \end{cases}$ where $\mathcal{D} = \mathbb{R}^2 \setminus \{ (x, 0) \mid x \le 0 \}$. Note reversal of arguments y, x compared with mathematical definition x, y.
$\texttt{acosh}(x)$	$\begin{cases} \texttt{trv} & \text{if } \boldsymbol{x} \not\subseteq [1, +\infty]; \\ \texttt{com} & \text{if } \boldsymbol{x} \subseteq [1, +\infty], \boldsymbol{x} \text{ bounded, and result bounded at Level 2;} \\ \texttt{dac} & \text{otherwise.} \end{cases}$
$\texttt{atanh}(x)$	$\begin{cases} \texttt{trv} & \text{if } \boldsymbol{x} \not\subseteq (-1, 1); \\ \texttt{com} & \text{if } \boldsymbol{x} \subseteq (-1, 1)\text{, and result bounded at Level 2;} \\ \texttt{dac} & \text{otherwise.} \end{cases}$

2. For a function defined by an expression, finding the strongest decoration over a box is typically hard in the same way and for the same reasons that finding the tightest interval enclosure of the exact range is hard. Straightforward interval evaluation usually does not find it. A trivial example is the expression $f(x) = \sqrt{x} - x$. As a real function it gives $f(x) = 0$, which is continuous for all x, so that $\operatorname{dec}(f, \boldsymbol{x}) = \texttt{dac}$ for any nonempty interval $\boldsymbol{x}$. But for any $\boldsymbol{x}$ of more than one point, evaluating $f(\boldsymbol{x})$ as in §10.6 gives $\texttt{trv}$ as the decoration, because it takes the square root of an interval containing negative points.

TABLE 2. Required forward elementary functions: step functions. The set of jump points is shown. The range and local decoration are computed by the algorithm below.

Function φ	Set J of jump points of φ.
$\texttt{sign}(x)$	$J = \{0\}$
$\texttt{ceil}(x), \texttt{floor}(x)$	$J = \mathbb{Z}$
$\texttt{trunc}(x)$	$J = \mathbb{Z} \setminus \{0\}$
$\texttt{roundTiesToEven}(x), \texttt{roundTiesToAway}(x)$	$J = \{\, n + \frac{1}{2} \mid n \in \mathbb{Z}\,\}$

At an infinite endpoint, the value of φ is taken as its limiting value, e.g. $\texttt{sign}(-\infty) = -1$, $\texttt{ceil}(+\infty) = +\infty$.

```
Input: nonempty $\boldsymbol{x} = [\underline{x}, \overline{x}]$, possibly unbounded.
$\underline{y} = \varphi(\underline{x})$, $\overline{y} = \varphi(\overline{x})$
if $\underline{y} = \overline{y}$
  if $\underline{x} \notin J$ and $\overline{x} \notin J$ and $\boldsymbol{x}$ is bounded
    $d = \texttt{com}$
  else
    $d = \texttt{dac}$
  end if
else
  $d = \texttt{def}$
end if
Output: range enclosure $[\underline{y}, \overline{y}]$ and local decoration $d$.
```

Similarly, the computed decoration of $f(x) = \sqrt{-1 - x^2}$ in the example above will be $\texttt{emp}$ or $\texttt{trv}$ for any $\boldsymbol{x}$, never the correct $\texttt{ill}$.

Note also that though $\texttt{trv}$ is trivial in itself, to have $\operatorname{dec}(f, \boldsymbol{x}) = \texttt{trv}$ is not trivial: it asserts $p_{\texttt{emp}}$ and $p_{\texttt{def}}$ are both false. The first implies that $\boldsymbol{x}$ has a point in $\operatorname{Dom}(f)$, the second that $\boldsymbol{x}$ has a point outside $\operatorname{Dom}(f)$; together these imply $\boldsymbol{x}$ is an interval of positive length.

3. Consider exact arithmetic DIE of $f(x, y) = \sqrt{x(y - x) - 1}$ with various input intervals $\boldsymbol{x}$, $\boldsymbol{y}$. Finite precision would produce valid but usually slightly different results. The natural domain $\operatorname{Dom}(f)$ is easily seen to be the union of the regions $x > 0$, $y \geq x + 1/x$ and $x < 0$, $y \leq x + 1/x$ in the plane.

(i) Let $\boldsymbol{x} = [1, 2]$, $\boldsymbol{y} = [3, 4]$, defining a box $(\boldsymbol{x}, \boldsymbol{y})$ contained in $\operatorname{Dom} f$. Applying the $\texttt{newDec}$ function gives initial decorated intervals $\boldsymbol{x}_{dx} = [1, 2]_{\texttt{dac}}$, $\boldsymbol{y}_{dy} = [3, 4]_{\texttt{dac}}$. The first operation is

$$\boldsymbol{u}_{du} = \boldsymbol{y}_{dy} - \boldsymbol{x}_{dx} \qquad = [1, 3]_{\texttt{dac}}.$$

Namely, subtraction is defined and continuous on all of $\mathbb{R}^2$, and bounded on bounded rectangles (call this property "nice" for short), so the bare result decoration is $du' = \operatorname{dec}(-, (\boldsymbol{y}, \boldsymbol{x})) = \texttt{dac}$, whence by (26) the decoration on $\boldsymbol{u}$ is $du = \min\{du', dy, dx\} = \min\{\texttt{dac}, \texttt{dac}, \texttt{dac}\} = \texttt{dac}$. Multiplication is also "nice", so the second operation similarly gives

$$\boldsymbol{v}_{dv} = \boldsymbol{x}_{dx} \times \boldsymbol{u}_{du} \qquad = [1, 6]_{\texttt{dac}}.$$

The constant 1, following §9.4.4, becomes a decorated interval function returning the constant value $[1, 1]_{\texttt{dac}}$. The next operation is again "nice", and gives

$$\boldsymbol{w}_{dw} = \boldsymbol{v}_{dv} - 1 \qquad = [0, 5]_{\texttt{dac}}$$

Finally $\sqrt{\cdot}$ is defined, continuous and bounded on $\boldsymbol{w} = [0, 5]$, so, arguing similarly, one has the final result

$$\boldsymbol{f}_{df} = \sqrt{\boldsymbol{w}_{dw}} \qquad = [0, \sqrt{5}]_{\texttt{dac}}.$$

By the FTDIA it is thus proven that for the box $\boldsymbol{z} = (\boldsymbol{x}, \boldsymbol{y}) = ([1, 2], [3, 4])$,

$$[0, \sqrt{5}] \supseteq \operatorname{Rge}(f \mid \boldsymbol{z}),$$
$$p_{\mathtt{dac}}(f, \boldsymbol{z})\text{holds.}$$

That is, f is defined, continuous and bounded on $1 \le x \le 2,\ 3 \le y \le 4$, and its range over this box is a subset of $[0, \sqrt{5}]$.

(ii) Let $\boldsymbol{x} = [1, 2]$ as before, but $\boldsymbol{y} = [\frac{5}{2}, 4]$. The box $\boldsymbol{z}$ is still contained in $\operatorname{Dom} f$ so the true value of $\operatorname{dec}(f, \boldsymbol{z})$ is still `dac`. However the evaluation fails to detect this because of interval widening due to the dependence problem of interval arithmetic. Namely after $\boldsymbol{u}_{du} = [\frac{5}{2}, 3]_{\mathtt{dac}}$, $\boldsymbol{v}_{dv} = [\frac{5}{2}, 6]_{\mathtt{dac}}$, $\boldsymbol{w}_{dw} = [-\frac{1}{2}, 5]_{\mathtt{dac}}$, the final result has interval part $\boldsymbol{f} = \sqrt{[-\frac{1}{2}, 5]} = [0, \sqrt{5}]$ as before, but $\sqrt{\cdot}$ is not everywhere defined on $\boldsymbol{w}$, so that $dw' = \operatorname{dec}(\sqrt{\cdot}, \boldsymbol{w}) = \operatorname{dec}(\sqrt{\cdot}, [-\frac{1}{2}, 5]) = \mathtt{trv}$ giving $\operatorname{dec}(\sqrt{\cdot}, \boldsymbol{w}_{dw}) = \min\{dw', dv\} = \mathtt{trv}$, so finally $\boldsymbol{f}_{df} = [0, \sqrt{5}]_{\mathtt{trv}}$. This is a valid enclosure of the decorated range $[0, \sqrt{5}]_{\mathtt{dac}}$, but we have been unable to verify the `dac` property.

(iii) If $\boldsymbol{x} = [1, 2]$, $\boldsymbol{y} = [1, 1]$, the box $\boldsymbol{z}$ is now wholly outside $\operatorname{Dom} f$, and evaluation detects this, giving the exact result $\boldsymbol{f}_{df} = \emptyset_{\mathtt{emp}}$. However, if $\boldsymbol{x} = [1, 2]$, $\boldsymbol{y} = [1, \frac{3}{2}]$, the box is still wholly outside $\operatorname{Dom} f$, but owing to widening, evaluation fails to detect this, giving $\boldsymbol{f}_{df} = [0, 0]_{\mathtt{trv}}$—a valid enclosure but of little use.

20. Implementation of compressed interval arithmetic

Table 3 gives tables of compressed arithmetic, §10.12, for the four basic operations. Here c, d are bare decorations less than the threshold τ, and $\boldsymbol{x}, \boldsymbol{y}$ are bare intervals. Independently of τ, if any input is the decoration `ill` the result is `ill`, else if any input is the interval $\emptyset$ the result is $\emptyset$. The tables below give the remaining cases where

$$\mathtt{trv} \le c < \tau,\ \mathtt{trv} \le d < \tau,\ \text{and } \boldsymbol{x},\ \boldsymbol{y} \text{ are nonempty.} \tag{42}$$

Table 3. Compressed interval operations for $+, -, \times, \div$ and $\sqrt{\cdot}$ with threshold $\tau \in \{\mathtt{trv}, \mathtt{def}, \mathtt{dac}, \mathtt{com}\}$.

Binary operations, where $\boldsymbol{x}$ or c is the left operand and $\boldsymbol{y}$ or d is the right operand.

$+, -, \times$	$\boldsymbol{y}$	d
$\boldsymbol{x}$	Normal bare interval result	d
c	c	$\min(c, d)$

$\div$	$\boldsymbol{y} = [0, 0]$	$0 \in \boldsymbol{y} \ne [0, 0]$	$0 \notin \boldsymbol{y}$	d
$\boldsymbol{x}$	`emp`	If τ>`trv` then `trv`, else normal bare interval result	Normal bare interval result	`trv`
c	`emp`	`trv`	c	`trv`

Square root, where $\boldsymbol{x} = [\underline{x}, \overline{x}]$.

	case	
$\sqrt{\boldsymbol{x}}$	$\overline{x} < 0$	`emp`
	$\underline{x} < 0 \le \overline{x}$	If τ>`trv` then `trv`, else normal bare interval result
	$\underline{x} \ge 0$	Normal bare interval result
$\sqrt{c}$		`trv`

Some examples of compressed arithmetic follow. In items (b) onwards, conditions (42) are assumed.

(a) Justification for $\mathtt{emp} + \boldsymbol{x} = \mathtt{emp}$, independent of τ.
This promotes to $(\emptyset, \mathtt{emp}) + (\boldsymbol{x}, \tau) = \big(\emptyset, \min(\mathtt{emp}, \tau, \mathtt{emp})\big)$. Since $\mathtt{emp} < \tau$ this equals $(\emptyset, \mathtt{emp})$ which gives an exception (again because $\mathtt{emp} < \tau$) so is recorded as the bare decoration emp. The same holds if $+$ is replaced by $-$, $\times$ or $\div$.
(b) Justification for $\boldsymbol{x} \times d = d$ independent of τ.
Since $\boldsymbol{x}$ is nonempty and $d \geq \mathtt{trv}$, this promotes to $(\boldsymbol{x}, \tau) \times (\boldsymbol{y}, d)$ with arbitrary nonempty $\boldsymbol{y}$, giving $\big(\boldsymbol{x} \times \boldsymbol{y}, \min(\tau, d, e)\big)$ where e is dac if $\boldsymbol{x} \times \boldsymbol{y}$ is bounded, otherwise def. Now $d < \tau$ so d cannot exceed def, hence $d \leq e$, so $\min(\tau, d, e) = d$.
(c) Justification for $c/d = \mathtt{trv}$ independent of τ.
Since $c, d \geq \mathtt{trv}$, c/d promotes to $(\boldsymbol{x}, c)/(\boldsymbol{y}, d)$ with arbitrary nonempty $\boldsymbol{x}, \boldsymbol{y}$, giving $\big(\boldsymbol{x}/\boldsymbol{y}, \min(c, d, e)\big)$ where $e = \mathtt{emp}$ if $\boldsymbol{y} = [0, 0]$, else $e = \mathtt{trv}$ if $0 \in \boldsymbol{y}$, else $e = \mathtt{dac}$. So $\min(c, d, e) \geq \mathtt{trv}$ and can equal trv, so the tightest enclosing decoration is trv.
(d) Justification for $\boldsymbol{x}/\boldsymbol{y}$ when $0 \in \boldsymbol{y} \neq [0, 0]$.
$\boldsymbol{x}/\boldsymbol{y}$ promotes to $(\boldsymbol{x}, \tau)/(\boldsymbol{y}, \tau)$ giving $\big(\boldsymbol{x}/\boldsymbol{y}, \min(\tau, \tau, \mathtt{trv})\big) = (\boldsymbol{x}/\boldsymbol{y}, \mathtt{trv})$. If $\tau > \mathtt{trv}$ this gives an exception so the decoration trv is returned; if $\tau = \mathtt{trv}$ it is not an exception, so the interval $\boldsymbol{x}/\boldsymbol{y}$ is returned.
(e) Justification for $\sqrt{\boldsymbol{x}}$ with $\boldsymbol{x} = [\underline{x}, \overline{x}]$ and $\underline{x} < 0 \leq \overline{x}$.
$\sqrt{\boldsymbol{x}}$ promotes to $\sqrt{(\boldsymbol{x}, \tau)}$, giving $(\sqrt{\boldsymbol{x}}, \mathtt{trv})$ which in the given case equals $([0, \sqrt{\overline{x}}], \mathtt{trv})$. As with the previous item, if $\tau > \mathtt{trv}$ then trv is returned; if $\tau = \mathtt{trv}$ then $\sqrt{\boldsymbol{x}}$ is returned.

21. Proofs of correctness for compressed interval arithmetic

⚠ To be completed.

22. The fundamental theorem of decorated interval arithmetic

We assume the seven-decoration set $\mathbb{D}$ of decorations defined by (17). However the proof of the fundamental theorem is largely independent of the particular set of decorations chosen.

It is necessary first to clarify the case of a zero-argument arithmetic operation φ, which represents a real constant. A point argument of a general k-ary φ is a tuple $u = (u_1, \ldots, u_k) \in \mathbb{R}^k$. For $k = 0$ this is the empty tuple $()$, which is the unique element of $\mathbb{R}^0$.

For an interval version, an argument for general k is $\boldsymbol{u} = (\boldsymbol{u}_1, \ldots, \boldsymbol{u}_k)$, representing the subset of $\mathbb{R}^k$ specified by k constraints $u_1 \in \boldsymbol{u}_1, \ldots, u_k \in \boldsymbol{u}_k$. For $k = 0$ there are no such constraints, so the input “box” to an interval version cannot be empty: it is always the whole of $\mathbb{R}^0 = \{()\}$.

However φ can have empty domain, in which case it is the “Not a Number” function NaN; otherwise its domain is $\mathbb{R}^0$ and it has a real value. Clearly, if φ is NaN then $p_{\texttt{ill}}(\varphi, \mathbb{R}^0)$ holds, otherwise $p_{\texttt{bnd}}(\varphi, \mathbb{R}^0)$ holds.

We now prove:

Theorem 22.1 (Fundamental Theorem of Decorated Interval Arithmetic, FTDIA)**.**
Let $\boldsymbol{f}_{df} = f(\boldsymbol{x})$ be the result of evaluating an arithmetic expression $f(z_1, \ldots, z_n)$ over a bare box $\boldsymbol{x} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n) \in \overline{\mathbb{IR}}^n$ using any decorated interval version f of f. Then in addition to the enclosure

$$\boldsymbol{f} \supseteq \operatorname{Rge}(f \,|\, \boldsymbol{x}) \tag{43}$$

given by Moore’s FTIA Theorem (page 10), we have

$$p_{df}(f, \boldsymbol{x}) \text{ holds.} \tag{44}$$

Proof. The case where $\boldsymbol{x}$ is empty is a special case. By case (Eval1) of the definition of a decorated interval version in §10.6, $\boldsymbol{f}_{df} = \emptyset_{\texttt{ein}}$. Also $\operatorname{Rge}(f \,|\, \boldsymbol{x}) = \emptyset$ and by definition, $p_{\texttt{ein}}(f, \boldsymbol{x})$ holds, so that (43, 44) hold.

Otherwise $\boldsymbol{x}$ is nonempty (so each of its components is nonempty) and we proceed by induction on the number of operations in f.

The base case, where this number is zero, is that f is a variable, say z_i. Then it defines the function $f(x) = x_i$. By case (Eval2) in §10.6, $\boldsymbol{f} = \boldsymbol{x}_i = \operatorname{Rge}(f \,|\, \boldsymbol{x})$ and df is such that $p_{df}(\mathrm{Id}, \boldsymbol{x}_i)$ holds, where Id is the identity function $\mathrm{Id}(x) = x$ on $\mathbb{R}$ (specifically, $df = \texttt{bnd}$ if $\boldsymbol{x}_i$ is bounded, and $df = \texttt{dac}$ otherwise). Then $p_{df}(f, \boldsymbol{x}) = p_{df}(\mathrm{Id}, \boldsymbol{x}_i)$ holds. Thus (43, 44) hold.

Otherwise $\boldsymbol{x}$ is nonempty and $f = \varphi(g_1, \ldots, g_k)$ where φ is an arithmetic operation of arity $k \geq 0$, and the g_i are expressions having fewer operations than does f. When f and the g_i are regarded as point functions, this means $f(x) = \varphi(g_1(x), \ldots, g_k(x))$ for $x \in \mathbb{R}^n$.

By the inductive hypothesis the theorem holds for each g_i. (The case $k = 0$, where φ is a real constant or NaN, needs no special treatment.) So (43, 44) applied to g_i give for $i = 1, \ldots, k$

$$\boldsymbol{g}_i \supseteq \operatorname{Rge}(g_i \,|\, \boldsymbol{x}), \tag{45}$$

$$p_{dg_i}(g_i, \boldsymbol{x}) \quad \text{holds.} \tag{46}$$

By the definition of a decorated interval version of f, $\boldsymbol{f}_{df}$ is computed using a decorated interval extension of φ, hence by the definition in §10.6,

$$\boldsymbol{f} \supseteq \operatorname{Rge}(\varphi \,|\, \boldsymbol{g}), \tag{47}$$

$$df = \min\{d\varphi, dg_1, \ldots, dg_k\} \tag{48}$$

for some $d\varphi$ such that

$$p_{d\varphi}(\varphi, \boldsymbol{g}) \text{ holds.} \tag{49}$$

We show first (43) and then (44). Denote here $u_k = g_i(x)$ for some $x \in \boldsymbol{x}$. Then

$$u_k = g_i(x) \in \operatorname{Rge}(g_i \,|\, \boldsymbol{x}) \subseteq \boldsymbol{g}_i. \tag{50}$$

For any $v \in \operatorname{Rge}(f \,|\, \boldsymbol{x})$, there is $x \in \boldsymbol{x}$ such that $v = f(x)$. Then, using (50, 47),

$$\begin{aligned} v = f(x) &= \varphi\big(g_1(x), \ldots, g_k(x)\big) \\ &= \varphi\big(u_1, \ldots, u_k\big) \\ &= \varphi(u) \in \operatorname{Rge}(\varphi \,|\, \boldsymbol{g}) \subseteq \boldsymbol{f}. \end{aligned}$$

Since v was arbitrary, this proves (43).

It remains to prove (44). Corresponding to the different meanings of the decorations, this is verified on a case by case basis, starting with the least decoration.

Case $df = \texttt{ill}$. Then either some $dg_i = \texttt{ill}$ or $d\varphi = \texttt{ill}$.

- If $dg_i = \texttt{ill}$, by (46) $\operatorname{Dom}\varphi$ is empty.
- If $d\varphi = \texttt{ill}$, hence by (49) $\operatorname{Dom}\varphi$ is empty.

In either case, by the definition of the point function f, $\operatorname{Dom} f$ is empty so (44) holds.

Case $df = \texttt{emp}$. Then either some $dg_i = \texttt{emp}$ or $d\varphi = \texttt{emp}$.

- If $dg_i = \texttt{emp}$, by (46) $\boldsymbol{x}$ is disjoint from $\operatorname{Dom} g_i$.
- If $d\varphi = \texttt{emp}$, by (49) $\boldsymbol{g}$ is disjoint from $\operatorname{Dom}\varphi$.

In either case there is no $x \in \boldsymbol{x}$ for which $f(x)$ is defined. That is, $\boldsymbol{x}$ is disjoint from $\operatorname{Dom} f$, and (44) holds.

Case $df = \texttt{trv}$. This is always true, and nothing needs to be shown.

Case $df = \texttt{def}$. Then each $dg_i \geq \texttt{def}$, and $d\varphi \geq \texttt{def}$. Thus by (46, 49), each g_i is everywhere defined on $\boldsymbol{x}$, with values in $\boldsymbol{g}_i$ by (45), and φ is everywhere defined on $\boldsymbol{g}$. Hence f is everywhere defined on $\boldsymbol{x}$ so again (44) holds.

Case $df = \texttt{dac}$. This is as the $\texttt{def}$ case with the addition that the restriction of each g_i to $\boldsymbol{x}$ is everywhere defined and continuous, and the restriction of φ to $\boldsymbol{g}$ is everywhere defined and continuous. Hence the restriction of f to $\boldsymbol{x}$ is everywhere defined and continuous so again (44) holds.

Case $df = \texttt{bnd}$. Then each $dg_i \geq \texttt{bnd}$, and $d\varphi \geq \texttt{bnd}$. By similar reasoning, the restriction of f to $\boldsymbol{x}$ is everywhere defined, continuous and bounded so again (44) holds.
(In fact all one needs to deduce this is that each $dg_i \geq \texttt{dac}$, and $d\varphi \geq \texttt{bnd}$.)

Case $df = \texttt{ein}$. This cannot occur since $\boldsymbol{x}$ was assumed nonempty.

Hence all cases have been covered. This completes the induction step and the proof. □

ANNEX E

Further material for set-based standard (informative)

This may possibly be included in some form in the Annexes to the standard.

23. Type conversion in mixed operations

⚠ This needs checking by language and compiler experts and should be the subject of a separate motion. Also, does it all belong in the decorations section? Indeed should it be in Level 1 at all?

Decorated interval arithmetic is designed for maximal safety, while being simple to handle by inexperienced users. Safety requirements can be enforced only by restrictions on the kinds of type conversions permitted.

Operations between integers and decorated intervals are well-defined and hence permitted, with integers treated as constant functions.

Operations between floats and decorated intervals are error-prone and hence forbidden, since, e.g., $(2/3) * \boldsymbol{x}$ in program text would generate uncovered roundoff, and $0.2 * \boldsymbol{x}$ would generate uncovered conversion errors. This ensures that the user must call explicitly a conversion function **iconst** that performs the outward rounding, see §13, to convey the precise semantics of such mixed expressions. This avoids a loss of containment because of rounding errors or conversion errors.

In particular, there is no implicit type casting for real times decorated interval. Therefore, $2/3*\boldsymbol{x}$ with reals or integers 2 and 3 and a decorated interval $\boldsymbol{x}$ results in a type error when trying to evaluate the multiplication.

However, implicit type casting for text constants times interval is harmless, as text constants have no arithmetic operations defined on them, hence they can be unambiguously type cast to decorated intervals when occurring in an interval expression if the implementation language allows that. Therefore, $2/3 * \boldsymbol{x}$ is allowed if the compiler translates 2 and 3 into constant functions.

Mixed operations between bare intervals and decorated intervals are also forbidden, to avoid loss of rigor through non-arithmetic operations; again, explicit conversion using the function newDec must be used. However, explicit, constant bare intervals in program code may be treated by the compiler as constant functions with uncertain value when the bare interval is nonempty, and as the ill-formed constant when the bare interval is empty or ill-formed.

24. The "Not an Interval" object

⚠ TO BE REVISED

From §9.4.4, a real scalar function with no arguments—a mapping $\mathbb{R}^n \to \mathbb{R}^m$ with $n = 0$ and $m = 1$—is a **real constant**.

This specification of constants gives a Level 1 definition of NaN, "Not a Number"—not as a value, but as a constant function. $\mathbb{R}^0$ is the zero-dimensional vector space $\{0\}$—it has one element, conventionally named 0. The real numbers c are in one-to- one correspondence with the mappings $c() : 0 \mapsto c$, so that $\mathbb{R}$ can be identified with the *total* functions $\mathbb{R}^0 \to \mathbb{R}$. There is one *non-total* $c()$, the function NaN() with empty domain and, therefore, no value.

From the definition in §9.4.3, an interval extension of a real constant with value c is any zero-argument interval function that returns an interval containing c. The *natural extension* returns the interval $[c, c]$.

Its natural interval extension is the constant interval function whose value is the empty interval.

the zero-argument function with empty domain is the real constant function with value NaN, "Not a Number". It is easily seen that NaN's natural interval extension is the interval constant function with value $\emptyset$, and its natural decorated interval extension is the decorated interval constant function with value $\text{NaI} = (\emptyset, \texttt{ill})$. (This was pointed out by Arnold Neumaier.)

The decorated interval NaI has behaviour that qualifies it for the role of "Not an Interval". By definition it signals that it is the result of evaluating a null function, with empty domain.

It is returned by any invalid call to an interval constructor, such as "the interval from 3 to NaN". It is unconditionally "sticky" within arithmetic expressions, in the sense that if any argument to an arithmetic operation is NaI, then that operation's output is NaI.

However, it cannot be generated "new" during evaluation of any expression that uses normal operations, even if the theoretical function being defined has empty domain. For example, the expression

$$f(x) = \sqrt{-1 - x^2}$$

clearly defines, over the reals, a function with empty domain; but decorated interval evaluation can *never* notice this. With any non-NaI input, it will return $(\emptyset, \texttt{emp})$ and not $(\emptyset, \texttt{ill})$.

Hence, in practice, NaI behaves as one expects it to do: it records the "taint of illegitimacy" of an interval's ancestry. A decorated interval is NaI iff it is the result of an ill-formed construction or is the computational descendant of such a result.

Bibliography

[1] Allen, James F. Maintaining knowledge about temporal intervals. Communications of the ACM 26, 832–843, (November 1983).

[2] Kulisch, Ulrich. Complete Interval Arithmetic and its Implementation on the Computer. Position paper, and the Dagstuhl 2008 proceedings.

[3] Kulisch, Ulrich. *Computer Arithmetic and Validity: Theory, Implementation, and Applications*. de Gruyter, Berlin, New York, (2008).

[4] Kulisch, Ulrich and Snyder, Van. *The exact dot product as basic tool for long interval arithmetic*. Position paper, P1788 Working Group, version 11, July 2009.

[5] Moore, Ramon E. *Interval Analysis*. Prentice-Hall, Englewood Cliffs, N.J., (1966).

[6] Nehmeier, Marco and Siegel, Stefan and Wolff von Gudenberg, Jürgen. Specification of hardware for interval arithmetic. Computing 94, 243-255, (2012).

[7] Neumaier, Arnold. Vienna Proposal for Interval Standardization. Faculty of Mathematics, University of Vienna, (December 2008). `http://www.mat.univie.ac.at/~neum`

[8] Pryce, John D. and Corliss, George F. Interval arithmetic with containment sets. Computing 78, 251–276, (2006).

[9] Pryce, John D. P1788 Motion 6: Multi-Format Support: Text and Rationale, (2009).