P1788/D9.2
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)

Introduction

This introduction is not part of IEEE Draft Standard P1788 for Interval Arithmetic.

This introduction explains some of the alternative interpretations, and sometimes competing objectives, that influenced the design of this 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 might 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 [6] 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 [9], 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 includes only the set-based flavor. Among other possible flavors are Kaucher/modal intervals; containment-sets; and the interval system of Siegfried Rump, which handles the relation between floating-point numbers and intervals, including overflow, in an elegant way, as well as being able to support open and half-open intervals. All of these extend classical IA in the defined sense.

Chapter 1 contains a common set of definitions and requirements that apply to all flavors; then the standard for each flavor is presented as a separate chapter. 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 Clause 5) 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, to consider 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 [**6**] 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 might 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, during program execution it can only be determined *after* evaluating an expression whether the conditions for any version of the FTIA hold; this is an important application of the standard's *decoration system*.

For each flavor included in the standard, its subdocument must state 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 (such as solving differential equations) where one must check the hypotheses of a *fixed point theorem* are satisfied:
   (a) one might 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 might 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 might 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 use 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 allows a rigorous check, for instance, that the conditions for applying a fixed-point theorem hold. 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 might 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 expert 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 interval 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 text strings.
- Interval-related exceptional conditions 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.
- How (for implementations supporting IEEE 754 arithmetic) operations act on the IEEE 754 status flags.
- 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 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");
– **comprise** indicates members of a set are exactly those objects having some property. 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. Clause 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 conform fully 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, 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.

In this standard, the units of transformation are called operations and 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 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 Clause 5) 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, *IEEE Standard for Floating-Point Arithmetic*. References in this document are to the 2008 revision.

3. Conformance Clause

3.1. Conformance overview. The P1788 Standard for Interval Arithmetic defines conformance for programming environments for interval arithmetic. A programming environment is a collection of processes for developing and executing computer programs. An *implementation*, in the following, is a programming environment that implements the Level 2 behavior of interval arithmetic as specified in this standard. The implementation may comprise a programming language as well as extensions in the form of libraries, classes, or packages that are necessary to satisfy the requirements for conformance. Requirements are given for the whole implementation; whether they are satisfied by the language itself or by an extension is irrelevant for conformance, see §1.7.

Conformance requirements are specified in this standard after the guidelines developed by OASIS, see [**11**]. In particular the OASIS concept of profiles is used to structure the requirements for a conforming implementation of a flavor of interval arithmetic.

A conforming implementation shall provide at least one of the following profiles, defined in Clause 7, called *standard flavors* of interval arithmetic in the context of this standard. A *profile* in this sense is a specific subset of functionality and requirements of this standard that may be provided by an implementation, but if it is provided shall be implemented exactly as specified in the corresponding section. A *sub-profile* is a nested profile. If the main profile is provided, an implementation may also support the sub-profile. If it does, it shall do so as specified.

An implementation may also provide additional flavors that shall conform to the core specification summarized in §7.1 and detailed in the following subclauses of Clause 7 as well as §8, 9.

[*Note. The procedure for submitting new flavors to be included into the standard is described in Annex A.*]

In any flavor of interval arithmetic, a *part* of an implementation is specified by a subset of the implementation's interval types (and where necessary to remove ambiguity, of its number formats). It comprises those types and formats, and associated operations of the implementation. It is a *conforming part* if it forms a conforming implementation of that flavor in its own right. Note this implies the full set of decorations of the flavor shall be supported.

The normative version of this standard is the English version. Translation to other languages is permitted.

3.2. Set-based interval arithmetic. An implementation of the set-based flavor, described in Chapter 2, shall satisfy the following requirements:

– provide the decorations specified in §11.2;
– support at least one bare interval type, see §12.5;
– derive each supported bare interval type from a number format with an associated rounding function, see §12.4, and provide an interval hull operation that maps arbitrary sets of real numbers to a corresponding interval, see §12.8;
– if multi-precision interval types are supported, define them as a parameterized sequence of interval types, see §12.7;
– provide implementations of the required operations in §12.12; required and recommended accuracies for these operations can be found in §12.10;
– if any of the recommended operations referenced in §12.13 are provided, provide them in a way that satisfies the requirements specified in the same clause; and
– provide input and output functions to convert intervals from and to strings as well as a public representation as specified in §13.2, §13.3, and §13.4; the string conversions shall satisfy containment in the general case and satisfy accuracy requirements for 754-conforming types as described in §13.2, 13.3, 13.4.

In all these, the implementation shall follow the representation rules defined in §14.1, essentially stating that booleans, strings, and decorations are represented as given in the references listed in the preceding bullets and that there is a one-to-one correspondence between the abstract number and interval formats in said references and the concrete formats used in the implementation.

As §14.2 states, each level 2 datum shall be represented by at least one level 3 object, and each level 3 object shall represent at most one level 2 datum.

3.2.1. *754-conformance.* In addition to basic conformance of the set-based flavor, an implementation may claim 754-conformance for all or part of the set-based flavor if the requirements of §12.6 are satisfied.

3.2.2. *Compressed decorated interval arithmetic.* An implementation may support the compressed arithmetic sub-profile of the set-based flavor. If compressed arithmetic is supported, it shall be as described in §11.10, in particular the implementation shall:

– provide an enquiry function to distinguish between intervals and decorations;
– provide a constructor for compressed intervals for each threshold value;
– provide a conversion function from compressed intervals to decorated intervals of the parent type;
– follow a *worst case semantic* for all arithmetic operations on compressed intervals; Clause B7 contains sample operation tables that satisfy this semantic; and
– provide compressed arithmetic implementations of the required operations in §12.12.

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

> [*Name of implementation and version*] is conforming to the P1788 Standard for Interval Arithmetic, version XX. It is conforming to the set-based flavor with 754-conformance for *list of 754-conforming supported interval types* and *with / without* compressed arithmetic. Additionally it provides *list of non-standard flavors.*

Part of the conformance claim shall be the completion of the following questionnaire.

3.4. Implementation conformance questionnaire.

(1) Implementation-defined behavior
    (a) What status flags or other means to signal the occurrence of certain decoration values in computations does the implementation provide if any, see Clause 8.1?

Does the implementation provide the **set-based flavor**? If so answer the following set of questions.

(2) Documentation of behavior
    (a) If the implementation supports implicit interval types, how is the interval hull operation realized? The answer may be given via an appropriate algorithm, see §12.8.
    (b) What accuracy is achieved (i.e., tightest, accurate, or valid) for each of the implementation's interval operations, see §12.10?
    (c) Under what conditions is a constructor unable to determine whether a Level 1 value exists that corresponds to the supplied inputs, see §12.12.7?
    (d) How are cases for rounding a Level 1 value to an $\mathbb{F}$-number handled that are not covered by the rules given in §12.12.8? This includes how the distance to an infinity is calculated when rounding a number? How are ties broken in rounding numbers if multiple numbers qualify as the rounded result?
    (e) How are interval datums converted to their exact text representations, see §13.4?

(3) Implementation-defined behavior
    (a) Does the implementation include the interval overlapping function, see §10.6.4? If so, how is it made available to the user?
    (b) Does the implementation store additional information in a NaI? What functions are provided for the user to set and read this information, see §11.3?
    (c) What means if any does the implementation provide for an exception to be signaled when a NaI is produced, see §11.3?
    (d) What interval types are supported besides the required ones, see §12.1.1?
    (e) What mechanisms of exception handling are used in exception handlers provided by the implementation, see §12.1.3? What additional exception handling is provided by the implementation?
    (f) What is the tie-breaking method used in rounding of supported number formats $\mathbb{F}$ that are not 754-conforming, see §12.12.8?
    (g) Does the implementation include different versions of the same operation for a given type and how are these provided to the user, see §12.12?
    (h) What combinations of formats are supported in interval constructors, see §12.12.7?

(i) What is the tightness of the result of constructor calls in cases where the standard does not specify it, see §12.12.7?
(j) What methods are used to read or write strings from or to character streams, see §13.1? Does the implementation employ variations in locales (such as specific character case matching)? This includes the syntax used in the strings for reading and writing.
(k) What is the tightness of the string to interval conversion for non-754-conforming interval types and the tightness for the interval to string conversion for all interval types, see §13.2, 13.3?
(l) What is the result of level 3 operations for invalid inputs, see §14.3?
(m) What are the interchange representations of the fields of the standard Level 3 representation listed in §14.4?
(n) What decorations does the implementation provide and what is their mathematical definition, see §8.2? How are these decorations mapped when converting an interval to the interchange format, see §14.4?
(o) What interchange formats if any are provided for non-754 interval formats and on non-754 systems, see §14.4? How are these provided to the user?

Does the implementation support the **compressed arithmetic** sub-profile of the set-based profile? If so answer the following set of questions.

(4) Implementation-defined behavior
(a) Which compressed interval types are provided, see §11.10?

Does the implementation provide **non-standard flavors** not defined in this standard, see §7.1? If so answer the following questions for each additional flavor.

(5) Flavor definition
(a) What is the set $\mathfrak{F}$ of intervals of the flavor? And what is the embedding map describing the relation to the common intervals, see §7.2?
(b) What decorations does the flavor provide, and what is their mathematical definition?
(c) What operations besides the required operations does the implementation provide, see §7.2? How are the $\mathfrak{F}$-versions of all provided operations—required and flavor-specific—defined, see §7.4?
(d) What interval types are provided for the intervals of $\mathfrak{F}$, see §7.5.1?
(e) What is the result of applying the flavor's `convertType` operation to a non-interval datum, see §7.5.2?
(f) For what interval types $\mathbb{T}$ of the flavor shall the implementation provide a $\mathbb{T}$-version of the flavor-specific operations, see §7.5.3? For what interval types should the implementation provide a $\mathbb{T}$-version?
(g) If possible, give a recommendation or requirements on the relation between the Level 2 number format $\mathbb{F}$ and the interval type $\mathbb{T}$ when passing from a Level 1 operation to a $\mathbb{T}$-version, see §7.5.3.
(h) How are exceptional cases in the evaluation of a $\mathbb{T}$-version of an operation handled, see §7.5.3?
(i) Does the flavor define accuracy modes in addition to valid and tightest? Where do these accuracy modes apply, see §7.5.4?

4. Notation, abbreviations, definitions

4.1. Frequently used notation and abbreviations.

754	IEEE-Std-754-2008 "IEEE Standard for Floating-Point Arithmetic".
IA	Interval 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.
$\mathbb{F}$	generic notation for a number format.
$\mathbb{T}$	generic notation for an interval type.
$\emptyset$, Empty	the empty set.
Entire	the whole real line.
NaI	Not an Interval.
NaN	Not a Number.
qNaN, sNaN	quiet and 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.
Val	map from member of concrete set to represented value in abstract set (e.g., from number format $\mathbb{F}$ to $\overline{\mathbb{R}}$).

4.2. Definitions.

The labels (AF) and (S) mark definitions that apply to all flavors, or the set-based flavor only, respectively.

4.2.1. **754 format.** (AF) A floating-point format that together with its associated operations conforms to IEEE-Std-754-2008. A **basic 754 format** is one of the five formats `binary32`, `binary64`, `binary128`, `decimal64`, `decimal128`.

4.2.2. **754-conforming implementation.** (S) An implementation of this standard, or a part thereof, that is built on a 754 system, uses only 754-conforming types, and satisfies the extra requirements for 754-conformance. Details in §12.6.

4.2.3. **754-conforming type.** (S) An inf-sup interval type derived from a 754 format, with its relevant operations; or the associated decorated type with its operations.

4.2.4. **754 system.** (AF) A programming environment, made up of hardware or software or both, that provides floating-point arithmetic conforming to IEEE-Std-754-2008.

4.2.5. **accuracy mode.** (AF) A way to describe the quality of an interval version of a function. See **tightness**, details in (AF) §7.5.4 and (S) §12.10.1.

4.2.6. **arithmetic operation.** (AF) A **version** of a **point operation**. Details in §6.1.

A **basic arithmetic operation** of IEEE-754 is one of the six functions $+$, $-$, $\times$, $\div$, fused multiply-add `fma` and square root `sqrt`.

4.2.7. **arity.** (AF) The number of arguments of a function. Details in §6.3.

4.2.8. **bare interval.** (AF) Same as **interval**; used to emphasize it is not decorated.

4.2.9. **bound.** (AF) One of the (extended-real) numbers $\underline{x}$, $\overline{x}$ for an interval that is, or has an interpretation as being, the set of numbers between $\underline{x}$ and $\overline{x}$, whether including or excluding the bounds themselves. (S) Details in §10.2.

4.2.10. **box.** (AF) 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. When an interval may be regarded as a set of points, it may be identified with the cartesian product $\boldsymbol{x}_1 \times \ldots \times \boldsymbol{x}_n$. (S) Details in §10.2.

4.2.11. **common evaluation.** (AF) Those evaluations of an operation (over common intervals) that have the same value in all flavors. Details in §7.3, Clause 9.

4.2.12. **common interval.** (AF) Those intervals that "modulo the embedding map" belong to all flavors. Details in §7.2.

4.2.13. **comparable.** (AF) A set of interval types is comparable if for any two of them, one is wider than the other, that is if, regarded as sets of mathematical intervals, they are linearly ordered by set inclusion.

4.2.14. **conforming part.** (AF) A subset (possibly the whole) of an implementation's types and formats, with associated operations, that forms a conforming implementation in its own right. In a multi-flavor implementation, it might be formed from a subset of the flavors. Details in §3.1.

4.2.15. **constant.** (AF) A **real constant** at Level 1 is defined to be a scalar point function with no arguments; this includes the constant NaN. Details in §6.1.

4.2.16. **contain.** (AF) A partial order relation between intervals that shall be defined in each flavor, and for common intervals shall coincide with normal set containment. Synonym **enclose**. Details in §7.2.

4.2.17. **datum.** (AF) One of the entities manipulated by finite precision (Level 2) operations of this standard. It can be a **boolean**, a **decoration**, a (bare) **interval**, a **decorated interval**, a **number** or a **string** datum. Number datums are organized into formats; bare and decorated interval datums are organized into types. An $\mathbb{F}$-datum or $\mathbb{T}$-datum means a member of the number format $\mathbb{F}$, or of the bare or decorated interval type $\mathbb{T}$. Details in Clause 5, §7.5.1, (S) §12.1.

4.2.18. **decoration.** (AF) One of the flavor-defined sets of values used to record events, called exceptional conditions, in interval operations: e.g., evaluating a function at points where it is undefined. Details in Clause 8.

(S) One of the five values `com`, `dac`, `def`, `trv` and `ill`. Details in Clause 11.

4.2.19. **decorated interval.** (AF) A pair (interval, decoration). Details in §7.5.1.

4.2.20. **domain.** (AF) 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 a point operation is part of its definition. E.g., the (point) operation of division x/y, in this standard, has arguments (x, y) in $\mathbb{R}^2$, and its domain is $\{ (x, y) \in \mathbb{R}^2 \mid y \neq 0 \}$. See also **natural domain**.

4.2.21. **elementary function.** (AF) Synonymous with **arithmetic operation**.

4.2.22. **enclose.** (AF) Synonymous with **contain**.

4.2.23. **exception, exceptional condition.** (AF) An **exception** is an event that occurs, and may be signaled, when an operation on some particular operands has no outcome suitable for every reasonable application. A signaled exception is handled in a language- or implementation-defined way. Details in §7.5.

An **exceptional condition** is one of the events handled by the decoration system; it is not an exception. Details in Clause 8.

(S) In the set-based standard, exceptions may occur in interval constructors (details in §12.1.3, 13.4) and in the `intervalPart` operation (details in §12.12.11).

4.2.24. **expression.** (AF) A symbolic form used to define a function. As used in this standard, it is a mathematical construct extracted from a section of run-time dataflow of a program, from which statements about enclosure of the range of a function may be deduced by using the FTIA. It may be represented by a computational graph, a code list or a normal algebraic expression. Details in Clause 6. An **arithmetic expression** is one whose operations are all arithmetic operations.

4.2.25. **explicit type.** (S) An interval type that has a uniquely defined interval hull operation.

4.2.26. **floating-point format.** (AF) A number format like those of 754, whose numbers have the form $x = s \times d_0.d_1 \ldots d_p \times b^e$ where integer $b \geq 2$ is the fixed radix, integer $p \geq 0$ is the fixed precision, $s = \pm 1$ is the sign, $d_0.d_1 \ldots d_p$ (a radix-b fraction) is the significand or mantissa, and e is an integer in a fixed exponent range $emin \leq e \leq emax$.

4.2.27. **fma.** (AF) Fused multiply-add operation, that computes $x \times y + z$. One of the basic arithmetic operations. Details in §9.1.

4.2.28. **format.** (Or **number format**.) (AF) One of the sets into which number datums are organized at Level 2, usually regarded as a finite set $\mathbb{F}$ of extended-reals with possibly also $-0, +0$.

If $\mathbb{F}$ is a format, an **$\mathbb{F}$-datum** means a member of $\mathbb{F}$, and an **$\mathbb{F}$-number** means a non-NaN member of $\mathbb{F}$. The **value** map Val maps $\mathbb{F}$-numbers to the extended-reals they denote: $\mathrm{Val}(-0) = \mathrm{Val}(+0) = 0$ and $\mathrm{Val}(x) = x$ for other $\mathbb{F}$-numbers.

(S) A format is **provided** if the implementation provides a representation of it, and **supported** if also it has properties specified in §12.4.

4.2.29. **FTIA.** (AF) The Fundamental Theorem of Interval Arithmetic. Details in §6.4. Refers to both the original versions that apply to bare intervals, and any decorated interval version applying to a flavor of this standard. For emphasis, the latter is sometimes referred to as the Fundamental Theorem of Decorated Interval Arithmetic (for that flavor).

4.2.30. **function.** Has the usual mathematical meaning of a partial function. Details in §6.1. Synonymous with **map**, **mapping**.

4.2.31. **hull.** (AF) (Or **interval hull**.) When not qualified by the name of an interval type, the hull of a subset $\boldsymbol{s}$ of $\mathbb{R}$ is the Level 1 hull, namely the tightest interval containing $\boldsymbol{s}$ in a flavor's sense of "contain". Details in §7.5.2.

(S) When $\mathbb{T}$ is an explicit type, the $\mathbb{T}$-hull of $\boldsymbol{s}$ is the unique tightest $\mathbb{T}$-interval containing $\boldsymbol{s}$.

When $\mathbb{T}$ is an implicit type, the $\mathbb{T}$-hull of $\boldsymbol{s}$ is a minimal $\mathbb{T}$-interval containing $\boldsymbol{s}$ as specified in the definition of the type.

4.2.32. **implementation.** (AF) When used without qualification, means a realization of an interval arithmetic conforming to the specification of this standard. Details in Clause 3.

4.2.33. **implicit type.** (S) An interval type that does not have a uniquely defined interval hull operation: the hull must be specified as part of the definition of the type.

4.2.34. **inf-sup.** (AF) Describes a representation of an interval based on its lower and upper bounds.

4.2.35. **interval.** (AF) At Level 1 the entities of the abstract data type forming one part of the interval model of an interval flavor. At Level 2 a $\mathbb{T}$-interval is a member of the bare interval type $\mathbb{T}$, or a non-NaI member of the decorated interval type $\mathbb{T}$. Details in §7.2, 7.5.1.

(S) A (bare) interval $\boldsymbol{x}$, see §10.2, is a closed connected subset of $\mathbb{R}$.

The set of all intervals is denoted $\overline{\mathbb{IR}}$. It comprises the empty set $\emptyset$ and the nonempty intervals $[\underline{x}, \overline{x}] = \{\, x \in \mathbb{R} \mid \underline{x} \le x \le \overline{x} \,\}$. Here $\underline{x} = \inf \boldsymbol{x}$ and $\overline{x} = \sup \boldsymbol{x}$, the **bounds** of the interval, are extended-real numbers satisfying $\underline{x} \le \overline{x}$, $\underline{x} < +\infty$ and $\overline{x} > -\infty$. Note $\pm\infty$ can be bounds of an interval but never members of it.

4.2.36. **interval extension.** (S) At Level 1, 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. 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 $\boldsymbol{x}$. Details in §10.4. For Level 2 interval extension, see §12.9.

A **decorated interval extension** of f is a function from decorated intervals to decorated intervals, whose interval part is an interval extension of f, and whose decoration part propagates decorations as specified in §11.6.

4.2.37. **interval vector.** (AF) See **box**.

4.2.38. **library.** (AF) In a given flavor, the set of Level 1 operations (Level 1 library) or those provided by an implementation (Level 2 library). Further classification may be made into the **point library**, **bare interval library** and **decorated interval library**. Details in §6.3, §9; see also §6.2.

4.2.39. **map, mapping.** See **function**.

4.2.40. **mathematical interval of constructor.** (AF) The arguments of an interval constructor, if valid, define a mathematical interval $\boldsymbol{x}$. The actual interval returned by the constructor is the tightest interval of the destination type that contains $\boldsymbol{x}$, see §12.12.7.

4.2.41. **mid-rad.** (AF) Describes a representation of an interval based on its midpoint and radius.

4.2.42. **NaI, NaN.** (AF) At Level 1 NaN is the function $\mathbb{R}^0 \to \mathbb{R}$ with empty domain. NaI is the natural interval extension of NaN, the map from subsets of $\mathbb{R}^0$ to subsets of $\mathbb{R}$ whose value is always the empty set. Details in §6.1.

(S) At Level 2 NaN is the Not a Number datum, which is a member of every supported number format. NaI is the Not an Interval datum, which is a member of every decorated interval type. Details in §11.3.

(AF) Other flavors may provide NaN and NaI. Details in §7.5.1.

4.2.43. **narrower.** See **wider**.

4.2.44. **natural domain.** (AF) For an arithmetic expression $\mathsf{f}(\mathsf{z}_1, \dots, \mathsf{z}_n)$, the natural domain is the set of $x = (x_1, \dots, x_n) \in \mathbb{R}^n$, where the expression defines a value for the associated point function $f(x)$, see Clause 6.

4.2.45. **no value.** (AF) A mathematical (Level 1) operation evaluated at a point outside its domain is said to have no value. Used instead of "undefined", which can be ambiguous. E.g., in this standard, real number division x/y has no value when $y = 0$. Details in §6.1.

4.2.46. **non-arithmetic operation.** (AF) An operation on intervals that is not an interval version of a point function; includes intersection and convex hull of two intervals. Details in §6.1.

4.2.47. **number.** (AF) 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**. Details in §10.1.

4.2.48. **number format.** See **format**.

4.2.49. **operation.** In a given flavor, synonymous with **supported function**.

4.2.50. **point function.** (AF) A mathematical (Level 1) function of real variables. Details in §6.1.

4.2.51. **primary version.** (AF) A Level 1 version of a generic function, from which all other versions are derived. Details in §6.1.

4.2.52. **provided operation.** (AF) In a given implementation of a flavor, an operation for which the implementation provides at least one Level 2 version; details in §6.1.

4.2.53. **range.** (AF) The range, $\mathrm{Rge}(f \mid \boldsymbol{s})$ of a function f over a set S is the set of values $f(x)$ at those points of S where f is defined. Details in §6.1.

4.2.54. **roundoff unit.** (AF) (Also called **machine epsilon**, **machine precision**.) In a number format $\mathbb{F}$, particularly a floating-point format, the smallest positive real number u such that $1 + u$ belongs to $\mathbb{F}$.

4.2.55. **string, text.** (AF) A **text string**, or just **string**, is a finite character sequence belonging to some alphabet, see §10.1. The term **text** is also used to mean strings generally; e.g., an operation having "numeric or text input" means each input is a number or a string.

4.2.56. **supported function.** In a given flavor, a function for which the flavor defines the primary version. Details in §6.1.

4.2.57. **tightest.** (AF) Smallest in the partial order of set containment, or in the "contains" relation of a flavor. The tightest set (unique, if it exists) with a given property is contained in every other set with that property.

Also denotes ones of the accuracy modes, see **tightness**.

4.2.58. **tightness.** (AF) The strongest accuracy mode (in (S) these are *tightest*, *accurate*, *valid*) that a given operation, in a given type, achieves for some input box, or uniformly over some set of inputs. Details in §7.5.4, 12.10.

4.2.59. **type.** (AF) (Or **interval type**.) One of the sets into which bare and decorated interval datums are organized at Level 2, usually regarded as a finite set $\mathbb{T}$ of Level 1 intervals, (a **bare type**) in the bare case; and of Level 1 decorated intervals together with the value NaI, (a **decorated type**) in the decorated case, see §12.5. Each bare type has a corresponding decorated type, and vice versa.

(S) A type is **provided** if the implementation provides a representation of it, and **supported** if also it has the properties specified in §12.5.

(AF) If $\mathbb{T}$ is a type, a **$\mathbb{T}$-datum** means a member of $\mathbb{T}$. A **$\mathbb{T}$-interval** for bare interval types means the same as a $\mathbb{T}$-datum, and for decorated types means a non-NaI member of $\mathbb{T}$.

4.2.60. **version.** (AF) A version of an operation is any of the actual operations denoted by a generic operation name. Details in §6.1.

4.2.61. **wider, narrower.** (AF) An interval type $\mathbb{T}'$ is wider than a type $\mathbb{T}$, and $\mathbb{T}$ is narrower than $\mathbb{T}'$, if $\mathbb{T}$ is a subset of $\mathbb{T}'$ when they are regarded as sets of Level 1 intervals, ignoring the type tags and possible decorations, see §12.5.1. Wider means having more precision, cf. 754 Definition 2.1.36.

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

TABLE 5.1. Specification levels for interval arithmetic

5. Structure of the standard in levels

For each flavor, the standard is structured into four levels, matching those defined in the 754 standard (754-2008 Table 3.1). They are summarized in Table 5.1.

Level 1, *mathematics*, defines the flavor's underlying theory. The entities at this level are mathematical intervals and operations on them. An implementation of the flavor shall implement this theory. In addition to an ordinary (bare) interval, this level defines a *decorated* interval, comprising a bare interval and a *decoration*. In all flavors, decorations implement the standard's way of handling exceptional conditions in interval operations.

Level 2, *discretization*, is the central part of the standard, approximating the mathematical theory by an implementation-defined finite set of entities and operations. A level 2 entity is called a *datum*[1]

Interval datums are organized into finite sets called *interval types*. An interval datum is a mathematical interval tagged by a symbol that indicates its type: an interval that "knows its type". The type abstracts a particular way of representing intervals (e.g., by storing their lower and upper bounds as IEEE `binary64` numbers). Most Level 2 arithmetic operations act on intervals of a given type to produce an interval of the same type.

Level 3 is about *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.

Level 4 is about *encoding* of interval objects into bit strings.

The Level 3 and 4 requirements in this standard are few, and mainly concern mappings from internal representations to external ones, such as interchange types.

The arrows in Table 5.1 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) Ignoring the type-tag, an interval datum *is* a mathematical interval.

(b) For each type $\mathbb{T}$, each mathematical interval has a unique interval datum as its $\mathbb{T}$-*hull*—a minimal enclosing interval of that type. This is with respect to a meaning of "contain". For the set-based flavor this is normal set inclusion, but in other flavors, e.g., Kaucher, might not always mean the same as set inclusion.

(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 might have more than one.

[1]Plural "datums" in this standard, since "data" is often misleading.

(*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 might have more than one.

[*Note. Items (c) and (d) are standard and necessary properties of representations. By contrast, the properties (a) and (b) of the maps from Level 1 to Level 2, and back, are fundamental design decisions of the standard.*]

6. Functions and expressions

6.1. Function definitions. In this document the terms **function**, **map** and **mapping** are synonymous and have the usual mathematical meaning (see §1.7 for general considerations) while **operation** has a specialized meaning. The following summarizes usage.

1. A (partial) **function** from a set X to a set Y associates to each point x in some subset of X, called the **domain** $D = \operatorname{Dom} f$ of f, a unique point $f(x)$ in Y called the value of f at x. At points outside D the function has no value. The notation $f : X \to Y$ means that f is a function from X to Y. A **total** function is one for which $D = X$.
2. The **range** of f over a subset S of X is the set $\operatorname{Rge}(f \mid S) = \{ f(x) \mid x \in S \cap \operatorname{Dom} f \}$. Points outside the domain are ignored—e.g., the range of the real square root function over $[-4, 4]$ is $[0, 2]$.

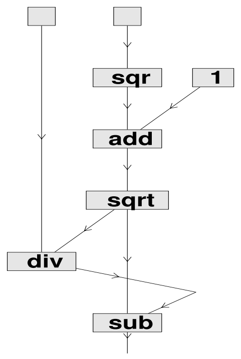
   Vector notation $f(x) = f(x_1, \ldots, x_n)$ may be used when X is a cartesian product $X = X_1 \times \cdots \times X_n$ with elements $x = (x_1, \ldots, x_n)$. The range may then be written $\operatorname{Rge}(f \mid S_1, \ldots, S_n)$ when $S = S_1 \times \cdots \times S_n$ with $S_j \subseteq X_j$ for each j.
3. A **point function** is a mathematical (Level 1) real function of real variables, $f : \mathbb{R}^n \to \mathbb{R}^m$ for some integers $n \geq 0$, $m > 0$.

   It is a **scalar** function when $m = 1$, otherwise a **vector-valued** function. When not otherwise specified, scalar is assumed. A function with $n = 0$ and $m = 1$—i.e., $\mathbb{R}^0 \to \mathbb{R}$—is a **real constant**. The unique such function with empty domain is by definition the **Not a Number** constant, NaN.
4. A **generic function** (more precisely, a “generic name” for functions) is a name that denotes any of several related functions called **versions** of the generic name. In each flavor a generic function has a Level 1 **primary version**—which may be the same in all flavors, or flavor-defined—from which all other versions are derived.

   E.g., `add` or “+” denotes mathematical addition of reals (the primary version), or Level 1 interval addition in some flavor, or finite precision (Level 2) addition in some interval type.
5. For a given flavor, a generic function is **supported**, and is an **operation**, if the flavor specifies its primary version; see §7.4. The operations in Clause 9 are so called because they are supported in all flavors.
6. For a given implementation of a flavor, an operation is **provided** if the implementation provides one or more Level 2 versions of it; see §7.5.3. Since such a version must be derived from a primary version, a function can only be provided if it is supported, i.e., if it is an operation.
7. An **arithmetic operation** is an operation whose primary version is a point function. This point function shall have nonempty domain (so Level 1 NaN cannot be a arithmetic operation). Any other operation is a **non-arithmetic operation**.
8. A **constructor** is a function that creates an interval from non-interval data (§9.4).

6.2. Expression definitions. An expression is some symbolic form that can be used to define a function—in general not a static object within program code, but derived dynamically from a particular program execution. Expressions are central to interval computation, because the Fundamental Theorem of Interval Arithmetic (FTIA) is about interpreting an expression in different ways:

(i) as defining a Level 1 point function f;
(ii) as defining various (depending on the finite precision interval types used) interval functions that give proven enclosures for the range of f over an input box $\boldsymbol{x}$;
(iii) as defining corresponding decorated interval functions that can give a stronger conclusion: e.g., in the set-based flavor, that f is everywhere defined, or everywhere continuous, on $\boldsymbol{x}$—enabling, say, an automatic check of the hypotheses of the Brouwer Fixed Point Theorem.



```
input  $v_{-1} = x_1$, $v_0 = x_2$
$v_1 = v_0^2$
$v_2 = v_1 + 1$
$v_3 = \sqrt{v_2}$
$v_4 = v_{-1}/v_3$
$v_5 = v_3 - v_4$
output  $y = v_5$
```

$$y = \sqrt{x_2^2+1} - \frac{x_1}{\sqrt{x_2^2+1}}$$

Computational graph (a) Code list (b) Algebraic expression (c)

FIGURE 6.1. Essentially equivalent notations for an expression. Notes:
In (a), structure is shown by labeling nodes with operations only; the order of arguments is shown by reading incoming edges left to right, e.g., the inputs to `sub` are the results of the preceding `sqrt` and `div`, in that order. Similarly the input nodes are x_1 and x_2 left to right.
Form (c) has redundancy in the sense of repeated subexpressions.
Constant 1 denotes the zero-argument constant function 1() wherever it occurs.

The meaning of (i) is flavor-independent; that of (ii) and (iii) is flavor-defined. The standard specifies behavior, at the individual operation level, that enables such conclusions, whether or not the notion "expression" exists in a programming language.

A *formal expression* defines a relation between certain mathematical variables—the *inputs*—and others—the *outputs*—via the application of named *operations*. It is by definition an acyclic (having an acyclic computational graph, see below) finite set of dependences between mathematical variables, defined by equations

$$v = \varphi(u_1, \dots, u_k), \tag{1}$$

where v and the u_i come from a set $\mathcal{X}$ of *variable-symbols*; φ comes from a set $\mathcal{F}$ of generic operations called the **library**; and distinct equations have distinct v's—the *single assignment* property.

An **arithmetic expression** is a formal expression, all of whose operations are arithmetic operations. Evaluating it using in turn the point version, an interval version and a decorated interval version of each of its operations, with appropriate inputs, gives the interpretations (i) to (iii) above, to which an FTIA appropriate to the flavor can be applied.

Non-arithmetic expressions (containing non-arithmetic operations) do not allow these three interpretations, e.g., the expression `mid(intersection(`$\boldsymbol{x}, \boldsymbol{y}$`))` in this standard can only be interpreted as a function with two interval inputs and a numeric output.

Three descriptions of an expression are outlined here. To apply the FTIA, it suffices to consider expressions that are *scalar*, with a single output.

(a) Drawing an edge from each u_i to v for each dependence-equation (1) defines the *computational graph* $\mathcal{G}$—Figure 6.1(a)—a directed graph over the node set $\mathcal{X}$. The dependences define an expression if and only if $\mathcal{G}$ is *acyclic*. There is then a nonempty set of *output* nodes having no outgoing edge, and a possibly empty set of *input* nodes having no incoming edge.

(b) Since $\mathcal{G}$ is acyclic, the equations can be ordered so that each one only depends on already known (input, or previously computed) values, thus representing the expression as a *code list*—Figure 6.1(b). In the notation of A. Griewank [**2**], the inputs are written $v_{1-n}, \dots, v_0$ where $n \ge 0$, conventionally given the aliases $x_1, \dots, x_n$, so x_i is the same as v_{i-n}. The operations are

$$v_r = \varphi_r(u_{r,1}, \dots, u_{r,k_r}), \quad (r = 1, \dots, m),$$

where $\varphi_r \in \mathcal{F}$ with arity (number of arguments) k_r, and each $u_{r,i}$ is a known v_j, that is $j = j(r,i) < r$. (Constants, which are operations of arity 0, may be referred to directly instead of assigned to a v_j.) Without loss, the outputs can be placed last so if there are p of them they are $v_{m-p+1}, \ldots, v_m$, aliased to $y_1, \ldots, y_p$, so that the code list defines a (vector) formal function $(y_1, \ldots, y_p) = f(x_1, \ldots, x_n)$; in case $p = 1$, it is written as a scalar function $y = f(x_1, \ldots, x_n)$.

Either m or n, but not both, can be zero. The case $n = 0$ and $m \geq 1$ gives a *constant expression*. For the scalar case $p = 1$, if $m = 0$ and $n = 1$ there are no operations, and y is the same as x_1, defining the *identity function* $y = f(x_1) = x_1$; while for $p = 1$, $m = 0$ and $n \geq 1$ there are n possibilities, the *coordinate projections* $y = \pi_j(x_1, \ldots, x_n) = x_j$ $(j = 1, \ldots, n)$.

(c) By allowing redundancy, an expression always can be converted to a normal (scalar) *algebraic expression*—Figure 6.1(c)—over the variable-set $\mathcal{X}$ and library $\mathcal{F}$, defined recursively as follows:
- if $x \in \mathcal{X}$ is a variable symbol, then x is an algebraic expression;
- if $\varphi \in \mathcal{F}$ is a function symbol of arity k and if e_i is an expression for $i = 1, \ldots, k$, then the *function symbol application* $\varphi(e_1, \ldots, e_k)$ is an algebraic expression.

Multiple outputs may be represented by tuples of separate algebraic expressions, e.g., the vector function $f(\theta) = (\cos(\theta), \sin(\theta))$. Redundancy may occur because this form cannot refer to a subexpression by name and must repeat it in full at each use, as with $\sqrt{x_2^2 + 1}$ in Figure 6.1(c).

The three forms are semantically equivalent, both at Level 1 and at Level 2. Because of its simple recursive definition, (c) is the form used in the FTIA proof in Clause B8.

When an expression is evaluated in interval mode, multiple instances of the same variable can lead to excessive widening of the final result: e.g., evaluating $x - x$ with an interval input $\boldsymbol{x}$ gives, not $[0, 0]$, but an interval twice the width of $\boldsymbol{x}$. The question, when it is valid to manipulate expressions (e.g., to replace $x - x$ by 0) is important for interval computation because of its potential to tighten enclosures, but is outside the scope of this standard.

6.3. Function libraries. An implementation's library for a flavor includes the following.
- The *Level 1 library* comprises the Level 1 operations, including those specified in Clause 9 and possible flavor-defined operations. These are the *primary versions* defined in §7.4 and, for arithmetic operations, their flavor-defined Level 1 bare and decorated interval versions.
- The *Level 2 library* comprises all Level 2 versions of the Level 1 operations using, e.g., different bare or decorated interval types or different number formats as the implementation may provide.

For arithmetic operations there is a further conceptual grouping of operations into the *point library* of primary versions at Level 1, and at Level 2 the *bare interval library* and the *decorated interval library*, to reflect different ways in which an arithmetic expression can be evaluated.

In this standard each generic operation $\varphi(u_1, \ldots, u_k)$, see eqn (1), is presented as having a fixed number k of arguments, termed its **arity**. Also for each version of φ at both Level 1 and Level 2, the **datatype** of each argument u_i, namely the set from which values of this argument are taken, is fixed.

At Level 1 the following datatypes are used by operations required in all flavors:
- *interval*, a generic term that maps to a Level 1 interval of a given flavor;
- *decoration*, a generic term that maps to a decoration of a given flavor;
- *number*, denoting a member of the reals $\mathbb{R}$ or extended reals $\overline{\mathbb{R}}$ depending on the flavor;
- *string*, a character sequence on a language- or implementation-defined alphabet;
- *boolean*, one of `false`, `true`.

Integer is used to describe parameterized sets of operations (e.g., `pown`() or array length of reduction operations), but is not considered a datatype in this standard. Other flavor-defined operations may use other datatypes, e.g., the 16-valued state of the Allen extended interval comparisons in §10.6.4 of the set-based flavor.

At Level 2, the decoration, string and boolean datatypes shall have the same meaning as a Level 1; each instance of the interval and number datatypes maps to one of various implementation-defined interval types and number formats, respectively.

Just as the standard is not concerned with the actual names or invocation methods, this conceptual view of libraries is independent of how the operations are presented in an actual computing environment: it could be via programming libraries, language primitives, infix operators, or other means. Also an implementation might permit the arity of an operation, or the datatype of any of its arguments, to be variable—possibly determined at run time. An implementation shall document how the formal operations are mapped to language entities.

6.4. The FTIA. The required arithmetic operations of Clause 9 are flavor-independent in the sense that the point version is the same in all flavors. Hence the point function $f(x) = f(x_1, \ldots, x_n)$, defined by an arithmetic expression made up of required operations, is the same in all flavors; in particular f has a flavor-independent *natural domain* $\mathrm{Dom}(f)$ determined by its constituent operations: the set of points $x \in \mathbb{R}^n$ where f *has a value* in the sense that the whole expression can be evaluated to give a real result.
[*Example. Knowing the definitions of division* x/y*, addition* $x + y$ *and square root* $\sqrt{x}$*, including that their domains are* $\{\,(x, y) \in \mathbb{R}^2 \mid y \neq 0\,\}$*, all of* $\mathbb{R}^2$*, and the interval* $0 \leq x < +\infty$*, respectively, one may deduce the natural domain of* $\sqrt{1 + 1/x}$ *is the union of intervals* $-\infty < x \leq -1$ *and* $0 < x < +\infty$*.*]

Each flavor has a version of Moore's fundamental theorem of interval arithmetic, FTIA: under suitable conditions, the range of such an f over a box is contained in the result of interval-evaluating the expression over the box. The details depend on the flavor's underlying theory, including its meaning of "contains". The standard is constructed so that for any conforming implementation of a conforming flavor, the flavor's FTIA holds in finite precision, not just at Level 1.

In the set-based flavor, Moore's theorem is as follows. The decoration system gives tools for checking the conditions for the "defined" and "continuous" forms during evaluation of a function.

Theorem 6.1 (Fundamental Theorem of Interval Arithmetic in set-based flavor)**.** *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) ("Basic" 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:*

$$\boldsymbol{y} \supseteq \mathrm{Rge}(f \mid \boldsymbol{x}) = \{\, f(x) \mid x \in \boldsymbol{x} \cap \mathrm{Dom}(f)\,\}. \tag{2}$$

(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* $\mathrm{Dom}(f) \supseteq \boldsymbol{x}$*.*

(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* $\boldsymbol{x}$*.*

(iv) ("Undefined" form of FTIA.) If some library operation in f *is nowhere defined on its inputs, while evaluating* $\boldsymbol{y}$*, then* f *is nowhere defined on* $\boldsymbol{x}$*, that is* $\mathrm{Dom}(f) \cap \boldsymbol{x} = \emptyset$*.*

6.5. Related issues. When program code contains conditionals (including loops), the run time data flow and hence the computed expression generally depends on the input data—for instance, Example (ii) in §11.8 where a function is defined piecewise. The user is responsible for checking that a property such as global continuity holds as intended in such cases. The standard provides no way to check this automatically.

Though the set operations `intersection` and `convexHull` are not point-operations and cannot appear directly in an arithmetic expression, they are useful for efficiently *implementing* interval extensions of functions defined piecewise, see the §11.8 example mentioned in the last paragraph.

The standard requires that at Level 2, for all interval types, all operations and all inputs other than NaI (if NaI is provided by the flavor), the interval part of a decorated interval operation equal the corresponding bare interval operation. This ensures that converting bare interval program code to use decorated intervals leaves the data flow entirely unchanged (provided no conditionals depend on decoration values, and NaI does not occur)—hence the computed expression and the interval part of its result are unchanged. If this were not so, there might in principle be an arbitrarily large discrepancy between the bare and the decorated versions of a computation that contains conditionals.

If a reproducible mode is supported, it should be supported in the same spirit as that of IEEE 754-2008: provided that certain programming restrictions (see §1.8 item (7)) are adhered to, the order of operations and the resulting values should be predictable based on the given input values.

7. Flavors

7.1. Flavors overview. The standard permits different interval behaviors via the flavor concept described in this clause. An **interval model** means a particular foundational approach to interval arithmetic, in the sense of a Level 1 abstract datatype of entities called intervals and of operations on them. A **flavor** is an interval model that conforms to the core specification described below.

A **provided flavor** is a flavor that the implementation provides in finite precision, see §7.5.

A **standard flavor** is one that has a specification in this standard, which extends the core specification. The set-based flavor is currently the only standard flavor. The procedure for submitting a new flavor for inclusion is described in Annex A.

An implementation shall provide at least one standard flavor. The implementation as a whole is conforming if each standard flavor conforms to the specification of that flavor, and each non-standard flavor conforms to the core specification.

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 over a procedure/function or a compilation unit.

An implementation with more than one flavor should provide means for conversion between suitably chosen pairs of interval types of different flavors. How this is provided—e.g., whether it is an import operation to, or an export operation from, the current flavor—is implementation-defined.

For brevity, phrases such as “A flavor shall provide, or document, a feature” mean that an implementation of that flavor shall provide the feature, or its documentation describe it.

The core specification of a flavor is summarized in the list below, and detailed in the following subclauses. The entities in the list are part of the Level 1 mathematical theory unless said otherwise.

(i) There is a flavor-independent set of *common intervals*, defined in §7.2.
(ii) There is a flavor-independent set of named *common operations*, defined in Clause 9.
(iii) There is a flavor-independent set of *common evaluations* of common operations, defined in §7.3. They have common intervals as input and, in the sense of §7.2, give the same result in any flavor.
(iv) There is a flavor-defined set of *intervals* of the flavor that, in the sense defined in §7.2, contains the common intervals as a subset. There is a flavor-defined finite set of *decorations* as described in Clause 8; in particular it includes the `com` decoration. A decorated interval is a pair (interval, decoration).
(v) There is a flavor-defined partial order called *contains* for the intervals, see §7.2, which for common intervals coincides with normal set containment.
(vi) A flavor shall define the Level 1 bare and decorated interval version of each of its arithmetic operations, and the Level 1 version (primary version) of each other operation, see §7.4.
(vii) At Level 2, intervals are organized into flavor-defined *types*. An (interval) type is essentially a finite set of Level 1 intervals; see §7.5.1.
(viii) The relation between a Level 1 operation and a version of it at Level 2, see §7.5.3, is summarized as follows. The latter evaluates the Level 1 operation on the Level 1 values denoted by its inputs. If (at those inputs) the operation has no value, an exception is signaled, or some default value returned, or both, in a flavor-defined way. Otherwise the returned value is converted to a Level 2 result of an appropriate Level 2 datatype. If the result is of interval type, overflow may occur in some flavors, causing an exception to be signaled.

7.2. Flavor basic properties. A flavor is described at Level 1 by a set $\mathfrak{F}$ of entities, the **intervals** of the flavor, a set of decorations, and a set of generic operations that includes the required operations of Clause 9. The symbol $\mathfrak{F}$ is also used to refer to the flavor as a whole.

The (flavor-independent) **common intervals** are defined to be the set $\mathbb{IR}$ of nonempty closed bounded real intervals used in classical Moore arithmetic [**6**].

The relation of $\mathfrak{F}$ to the common intervals is described by a one-to-one **embedding map** $\mathfrak{f} : \mathbb{IR} \to \mathfrak{F}$. Usually, $\mathfrak{f}(\boldsymbol{x})$ is abbreviated to $\mathfrak{f}\boldsymbol{x}$. The set of all $\mathfrak{f}\boldsymbol{x}$ for $\boldsymbol{x} \in \mathbb{IR}$ forms the **common intervals of** $\mathfrak{F}$. Usually $\boldsymbol{x}$ is identified with $\mathfrak{f}\boldsymbol{x}$ and $\mathbb{IR}$ is treated as a subset of $\mathfrak{F}$, for every flavor. To emphasize that this has been done, a statement may be said to hold *modulo the embedding map*.

The set of decorations of $\mathfrak{F}$ is finite, and includes the `com` decoration; see Clause 8.

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

]

For each flavor $\mathfrak{F}$, a relation $\supseteq$, called **contains** or **encloses**, shall be defined between intervals. It shall be a partial order: $\boldsymbol{x} \supseteq \boldsymbol{y}$ and $\boldsymbol{y} \supseteq \boldsymbol{z}$ imply $\boldsymbol{x} \supseteq \boldsymbol{z}$, and $\boldsymbol{x} = \boldsymbol{y}$ if and only if both $\boldsymbol{x} \supseteq \boldsymbol{y}$ and $\boldsymbol{y} \supseteq \boldsymbol{x}$ hold. For common intervals it shall have the normal meaning modulo the embedding map: if $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{IR} \subseteq \mathfrak{F}$ then $\boldsymbol{x} \supseteq \boldsymbol{y}$ in $\mathfrak{F}$ if and only if $\boldsymbol{x} \supseteq \boldsymbol{y}$ in the sense of set containment.
[*Example. In the Kaucher flavor,* (a,b) *is defined to contain* (a',b') *if and only if* $a \le a'$ *and* $b \ge b'$*; e.g.,* $[1,2] \supseteq [2,1]$ *is true. For common (proper) intervals, this coincides with normal containment.*]

7.3. Common evaluations.

For each Level 1 version of an operation φ for which at least one input or output is of bare interval datatype, a set of k-tuples $\boldsymbol{x} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k)$ is specified for which $\varphi(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k)$ shall have the same Level 1 value $\boldsymbol{y}$ in all flavors. Then $\boldsymbol{x}$ is a **common input**, and $\boldsymbol{y}$ is the **common value** of φ at $\boldsymbol{x}$. The $(k+1)$-tuple $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k; \, \boldsymbol{y})$ is a **common evaluation** of φ, alternatively written $\boldsymbol{y} = \varphi(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k)$. It may be called a common evaluation **instance** to emphasize that a specific input tuple is involved.

A tuple $\boldsymbol{x} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k)$ whose components $\boldsymbol{x}_i$ are common intervals—nonempty closed bounded intervals in $\mathbb{R}$—can be regarded as a nonempty closed bounded box in $\mathbb{R}^k$. A tuple whose $\boldsymbol{x}_i$ are common intervals of some flavor $\mathfrak{F}$ can be identified with such a box, modulo the embedding map.

There are two cases:

(a) If φ is one of the arithmetic operations in §9.1, its point version is a real function $y = \varphi(x_1, \ldots, x_k)$ of $k \ge 0$ real variables[2]. The common inputs are those boxes $\boldsymbol{x} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k)$ such that each $\boldsymbol{x}_i$ is common and the point version of φ is defined and continuous at each point of $\boldsymbol{x}$. The common value $\boldsymbol{y}$ is the range

$$\boldsymbol{y} = \operatorname{Rge}(\varphi \,|\, \boldsymbol{x}) = \{\, \varphi(x_1, \ldots, x_n) \mid x_i \in \boldsymbol{x}_i \text{ for each } i \,\}, \qquad \text{(here } \varphi \text{ is the point version).} \tag{3}$$

By theorems of real analysis, $\boldsymbol{y}$ is nonempty, closed, bounded and connected, so it is a common interval.

(b) If φ is one of the non-arithmetic operations in §9.2 to §9.6, it has a direct definition unrelated to a point function. Its common inputs comprise those $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k)$ such that each $\boldsymbol{x}_i$ that is of interval datatype is a common interval, and $\boldsymbol{y} = \varphi(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k)$ is defined and, if of interval datatype, is a common interval. The common value is $\boldsymbol{y}$.

7.4. Primary versions and Level 1 interval versions.

Let $\mathfrak{F}$ be a flavor. With terms as defined in §6.1, a primary version shall be **fully specified**, which means that the following are defined, whether in Clause 9 or by the flavor: the arity k of the function φ, namely the the number of arguments; the Level 1 datatype of each argument; the **domain**, namely the set of input tuples $(x_1, \ldots, x_k)$, with x_i of the specified datatypes, at which the operation has a value; and its value $y = \varphi(x_1, \ldots, x_k)$ at each such point. Equivalently, the flavor defines the primary version's function-graph: the set of all $(x_1, \ldots, x_k, y)$ such that $y = \varphi(x_1, \ldots, x_k)$.

Full specification makes it possible to test objectively whether, in a flavor $\mathfrak{F}$, an implemented Level 2 version of an operation encloses the Level 1 result (if an interval), approximates it (if numeric) or equals it (if boolean), whenever the Level 1 result is defined in $\mathfrak{F}$.

The possible cases for a generic operation φ are listed in the following subclauses.

[2] $\texttt{pown}(x,p) = x^p$ is treated as a family of functions $\varphi_p(x)$ of real x, parameterized by the integer p. Similarly for any function having some non-real arguments.

7.4.1. *Arithmetic operations.* Here the primary version is a point function, whether flavor-defined, or required in all flavors according to Clause 9.

The flavor shall define a mapping that takes an arbitrary point function $y = \varphi(x_1, \dots, x_k)$ to a function $\boldsymbol{y} = \varphi(\boldsymbol{x}_1, \dots, \boldsymbol{x}_k)$, called the **Level 1 (bare) interval version** of φ in $\mathfrak{F}$, for which the $\boldsymbol{x}_i$ and $\boldsymbol{y}$ are Level 1 bare intervals of $\mathfrak{F}$. Its evaluations shall include the common evaluations of §7.3, modulo the embedding map.

The flavor shall also define a mapping that takes such a point function to a function $\boldsymbol{Y} = \varphi(\boldsymbol{X}_1, \dots, \boldsymbol{X}_k)$, called the **Level 1 decorated interval version** of φ in $\mathfrak{F}$, for which the $\boldsymbol{X}_i$ and $\boldsymbol{Y}$ are Level 1 decorated intervals of $\mathfrak{F}$. If $\boldsymbol{y} = \varphi(\boldsymbol{x}_1, \dots, \boldsymbol{x}_k)$ is a common evaluation of the bare interval version, then $\boldsymbol{y}_{\texttt{com}} = \varphi((\boldsymbol{x}_1)_{\texttt{com}}, \dots, (\boldsymbol{x}_k)_{\texttt{com}})$ shall be an evaluation of the decorated interval $\mathfrak{F}$-version. It is termed a Level 1 **decorated common evaluation** of φ, and $((\boldsymbol{x}_1)_{\texttt{com}}, \dots, (\boldsymbol{x}_k)_{\texttt{com}})$ is a Level 1 **decorated common input**. These mappings from point function to Level 1 bare and decorated interval version shall ensure the validity of suitable flavor-defined Fundamental Theorems of interval arithmetic, and of decorated interval arithmetic, respectively.

The Level 1 interval versions shall be fully specified but may have no value for some combinations of input intervals.

[*Example. In the set-based flavor, the Level 1 bare interval version of φ is the* natural interval extension *of the point function, and is defined for all combinations of input intervals. However in classical Moore arithmetic and in Kaucher arithmetic, the Level 1 interval version of division $\boldsymbol{x}/\boldsymbol{y}$, for instance, is not defined when $0 \in \boldsymbol{y}$.*]

7.4.2. *Nonarithmetic operations required in all flavors.* For a nonarithmetic operation required in all flavors according to Clause 9, the primary version in $\mathfrak{F}$ shall be a Level 1 function for which each interval input or output becomes an interval of $\mathfrak{F}$, and whose evaluations include the common evaluations specified in Clause 9, modulo the embedding map.

[*Example. The interval constructors* `numsToInterval` *and* `textToInterval` *are required in all flavors. A flavor will in general extend these beyond the common evaluations. E.g., in the set-based flavor* `numsToInterval`(l, u) *constructs unbounded intervals by allowing non-common inputs with $l = -\infty$ and/or $u = +\infty$; in a Kaucher flavor it might construct improper intervals by allowing $l > u$.*]

7.4.3. *Flavor-defined nonarithmetic operations.* Besides the operations required in all flavors, a flavor may define *flavor-specific* operations that are required in each implementation of that flavor. Each such operation shall be fully specified at Level 1.

[*Examples.*

- *The set-based flavor defines reduction operations, one of which is the dot-product. The primary version of this is $s =$ `dot`$(x, y) = \sum_{i=1}^{n} x_i y_i$ where x, y are two real vectors of length n. Though this may be regarded as a point function (or a family of them, parameterized by n), it is not an arithmetic operation because it is not intended to have an interval version.*
- *A flavor might define a decorated interval version of a non-arithmetic operation.*
- *A flavor might define an operation on decorations, adapted to its decoration model.*
- *A flavor might provide other constructors besides the required ones, or give extra inputs to the required constructors. E.g., in a flavor that provides open and half-open as well as closed intervals, optional extra input(s) to* `numsToInterval` *might specify which bounds are open.*

]

[*Note. Apart from the decorated common evaluations the relation between the bare and decorated interval versions of an operation φ is flavor-defined, but barring special cases such as "Not an Interval" input, the interval part of the latter should equal the former. That is, if φ is defined at decorated interval inputs $(\boldsymbol{X}_1, \dots, \boldsymbol{X}_k)$, with decorated interval value $\boldsymbol{Y}$, and if the interval parts of $\boldsymbol{X}_i$ and $\boldsymbol{Y}$ are $\boldsymbol{x}_i$ and $\boldsymbol{y}$, then $\varphi(\boldsymbol{x}_1, \dots, \boldsymbol{x}_k)$ should in most cases be defined and have value $\boldsymbol{y}$.*]

7.5. The relation of Level 1 to Level 2.

Let $\mathfrak{F}$ be a flavor.

7.5.1. *Types.* At Level 2, bare intervals of $\mathfrak{F}$ shall be organized into finite sets called (bare interval) **types**. A type is an abstraction of a particular way to represent intervals. What types are provided by an implementation is both flavor-defined and language- or implementation-defined.

There shall be a one-one correspondence between bare interval types and decorated interval types (*bare types* and *decorated types* for short). An element of a decorated type is a pair $(\boldsymbol{x}, dx)$

where $\boldsymbol{x}$ belongs to the corresponding bare type, and dx belongs to the finite set of decorations of the flavor.

An implementation shall ensure that the decoration `com` is applied only to common intervals; a common interval may however have a decoration other than `com`.

Level 2 entities are called **datums**. A type may contain some exceptional datums, e.g., a "Not an Interval" datum, besides those that denote intervals. A **$\mathbb{T}$-datum** means a general member of a type $\mathbb{T}$; a **$\mathbb{T}$-interval** means a $\mathbb{T}$-datum that denotes an interval. If the type has no exceptional datums, these terms are synonymous.

Each Level 2 bare interval shall denote a unique Level 1 interval $\boldsymbol{x}$ of $\mathfrak{F}$ and is normally regarded as being that interval. However $\mathfrak{F}$ may formally define it as a pair $(\boldsymbol{x}, t)$ where t (the type name) is a symbol that uniquely identifies the type, thus making datums of different types different even if they denote the same Level 1 interval.

Hence, each Level 2 decorated interval denotes a unique Level 1 decorated interval of $\mathfrak{F}$ but may formally be a triple $(\boldsymbol{x}, dx, t)$ of interval, decoration and type name.

7.5.2. *Hull.* Each bare type $\mathbb{T}$ has a **$\mathbb{T}$-hull** operation. At Level 1 it shall be defined by an algorithm that maps an arbitrary interval $\boldsymbol{x}$ of $\mathfrak{F}$ to a minimal $\mathbb{T}$-interval $\boldsymbol{y}$ such that $\boldsymbol{y} \supseteq \boldsymbol{x}$ in the flavor's "contains" order, if such a $\boldsymbol{y}$ exists, and otherwise is undefined. Minimal means that if $\boldsymbol{z}$ is another $\mathbb{T}$-interval and $\boldsymbol{y} \supseteq \boldsymbol{z} \supseteq \boldsymbol{x}$ then $\boldsymbol{y} = \boldsymbol{z}$.

At Level 2, an implementation should provide the $\mathbb{T}$-hull for each supported bare type $\mathbb{T}$, as an operation `convertType` that maps an arbitrary interval of any other supported bare type of $\mathfrak{F}$ to its $\mathbb{T}$-hull if this exists, and signals the flavor-independent `IntvlOverflow` exception otherwise. Its action on non-interval datums is flavor-defined.

7.5.3. *Level 2 operations.* For each bare or decorated interval version of an arithmetic operation in §9.1 and for each operation in §9.2 to §9.6, a $\mathbb{T}$-*version* of φ shall be provided for each bare interval type $\mathbb{T}$ of the implementation of $\mathfrak{F}$. For flavor-defined operations the flavor shall specify whether an implementation shall or should provide a $\mathbb{T}$-version.

When passing from a Level 1 operation to a $\mathbb{T}$-version—whether required in all flavors or flavor-specific:

– Inputs or output of bare interval datatype change to type $\mathbb{T}$.
– Those of decorated interval datatype change to the decorated type of $\mathbb{T}$.
– Those of real datatype change to a type-dependent Level 2 number format $\mathbb{F}$. The flavor should make recommendations or requirements on the relation between $\mathbb{F}$ and $\mathbb{T}$.

In the description below, sometimes a *flavor-defined Level 2 value* may be returned in cases where no Level 1 value exists, instead of or alongside signaling an exception.
[*Example. For the* `mid`() *function at Level 2 in the set-based flavor, returning the midpoint of* Empty *as* NaN, *and of* Entire *as* 0, *illustrate flavor-defined values. Both values are undefined at Level 1. It was considered that no numeric value of* `mid`($\emptyset$) *makes sense, but that some algorithms are simplified by returning a default value* 0 *for* `mid`(Entire).]

For some operations and inputs the implementation might be unable to effectively compute some value or determine whether some statement is true or false—e.g., owing to constraints on time, space or algorithmic complexity. Below, the term *is found* means that the implementation is able to compute such a value or determine such a fact; *is not found* means the opposite.
[*Examples.*
– A constructor in the set-based flavor needs to ensure $l \le u$ *when forming an interval* $[l, u]$. *This may be hard to check, for some implementations and types.*
– In a flavor where it is a fatal error to evaluate an arithmetic operation outside its domain, it may be hard to decide if the required operation $\varphi(x) = \tan(x)$ *is defined on* $\boldsymbol{x} = [\underline{x}, \overline{x}]$ *when* $\underline{x}, \overline{x}$ *are large and differ by less than* π. *Let format* $\mathbb{F}$ *be IEEE754* `binary64` *and* $\mathbb{T}$ *the inf-sup type based on* $\mathbb{F}$. *The integer* $a = 214112296674652$ *differs from* $136308121570117\pi/2$ *by less than* 2.6e–16; *and* $a-1, a, a+1$ *are exact* $\mathbb{F}$*-numbers. One of the* $\mathbb{T}$*-intervals* $[a-1, a]$ *and* $[a, a+1]$ *(but not both) contains a singularity of* φ*; but which?*]

Evaluation of a $\mathbb{T}$-version $\varphi_{\mathbb{T}}$ of an operation φ shall be as if the following is done. Let $X = (X_1, \ldots, X_k)$ be the tuple of Level 1 values denoted by the inputs to $\varphi_{\mathbb{T}}$.

1. If the Level 1 operation is found to be undefined at X then $\varphi_{\mathbb{T}}$ signals the flavor-independent `UndefinedOperation` exception, or returns a flavor-defined value, or both. This includes the case where some input has no Level 1 value, e.g., a numeric input is NaN.
2. If it is not found whether the Level 1 operation is defined at X then $\varphi_{\mathbb{T}}$ signals the flavor-independent `PossiblyUndefinedOperation` exception, or returns a flavor-defined value, or both.
3. Otherwise, it is found that the Level 1 value $Y = \varphi(X_1, \dots, X_k)$ is defined in $\mathfrak{F}$.
   (a) If Y is a bare interval $\boldsymbol{y}$ there are two cases.
   – If a $\mathbb{T}$-interval $\boldsymbol{z}$ containing $\boldsymbol{y}$ is found (in particular if Entire exists in $\mathfrak{F}$ and is a $\mathbb{T}$-interval), then $\varphi_{\mathbb{T}}$ returns such a $\boldsymbol{z}$.
   – Otherwise, $\varphi_{\mathbb{T}}$ signals the flavor-independent `IntvlOverflow` exception and the returned result, if any, is flavor-defined.
   (b) If Y is a decorated interval $\boldsymbol{y}_{dy}$ there are three cases.
   – If φ is an arithmetic operation, and $(X_1, \dots, X_k)$ is a decorated common input (this implies $dy =$ `com`, see §7.4.1), and a common $\mathbb{T}$-interval $\boldsymbol{z}$ containing $\boldsymbol{y}$ is found, then $\varphi_{\mathbb{T}}$ returns such a $\boldsymbol{z}$ with the decoration `com`.
   – Otherwise, if a $\mathbb{T}$-interval $\boldsymbol{z}$ containing $\boldsymbol{y}$ is found (in particular if Entire exists in $\mathfrak{F}$ and is a $\mathbb{T}$-interval), then $\varphi_{\mathbb{T}}$ returns such a $\boldsymbol{z}$ with a flavor-defined decoration.
   – Otherwise, no such $\boldsymbol{z}$ is found. Then $\varphi_{\mathbb{T}}$ signals the `IntvlOverflow` exception and the returned result, if any, is flavor-defined.
   (c) If Y is numeric, $\varphi_{\mathbb{T}}$ returns a value of an appropriate number format, approximating Y in a type- and operation-dependent way.
   (d) If Y is boolean, $\varphi_{\mathbb{T}}$ returns Y.

7.5.4. *Measures of accuracy.* Two **accuracy modes** for an interval-valued operation are defined for all flavors. An accuracy mode is in the first instance a property of an individual evaluation of an operation φ, over an input box $\boldsymbol{x}$, returning a result of type $\mathbb{T}$. If the evaluation does not have an interval value at Level 1, its accuracy mode is undefined. The basic property of enclosure in the sense of the flavor, defined in §7.5.3 and required for conformance, is called *valid* accuracy mode. The property that the result equals the $\mathbb{T}$-hull of the Level 1 value is called *tightest* accuracy mode.

A flavor may define other accuracy modes, and may make requirements on the accuracy achieved by an implementation. To simplify the user interface, modes should be linearly ordered by strength, with *tightest* the strongest and *valid* the weakest, where mode M is stronger than mode M' if M implies M'.

7.6. Flavors and the Fundamental Theorem (informative). For a common evaluation of an arithmetic expression, each library operation is, modulo the embedding map, defined and continuous on its inputs so that it satisfies the conditions of the strongest, "continuous" form of the FTIA in Theorem 6.1. At Level 1, the range enclosure obtained by a common evaluation is 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 classical Kaucher flavor does not; conversely, Kaucher intervals have an extended FTIA 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.

The function f is assumed to be represented by an expression in the sense of Clause 6. 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. For example, in the set-based flavor, 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 might 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.

This standard's decoration system—in contrast with 754's way of reporting on the outcome of an operation—has no status flags. It provides a fully local handling of exceptional conditions in interval calculations—important in a concurrent computing environment. 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.

The following language- or implementation-defined features may be provided: (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 might 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.*]

8.2. Decoration definition and propagation. Each flavor shall document its set of provided decorations and their mathematical definitions. These are flavor-defined, except for 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 might convey purely Level 1 information—e.g., that φ is everywhere continuous on the box $\boldsymbol{x} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k)$. It might 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 might 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 initial decorations that lead to the most informative output-decoration. A flavor should provide a function that gives this initial decoration to a bare interval.

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.

Here φ denotes a Level 2 version of a point library arithmetic operation, see §7.5.3.

> If each input to φ, and the result, is common, and if each input is decorated `com`, then the result shall be decorated `com`.

[*Note.* `com` *is the only decoration that shall have the same meaning in all flavors.*

Informally, it records that the individual operation φ took bounded nonempty input intervals and produced a bounded (necessarily nonempty) output interval. This can be interpreted as "overflow did not occur". The propagation rule ensures that if the initial inputs to an arithmetic expression f *are common, and are initially decorated* `com`*, then the final result $\boldsymbol{y} = f(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k)$ is decorated* `com` *if and only if the evaluation of the whole expression was common as defined in §7.5.3.*]

[*Examples. Reasons why an individual evaluation of φ with common inputs $\boldsymbol{x} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k)$ might* 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 $\mathbb{T}$-datum $[0, \mathtt{REALMAX}]$. Then $\boldsymbol{x} + \boldsymbol{x}$ cannot be enclosed in a common $\mathbb{T}$-datum.*
– *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 bounds happens to be very close to a singularity of $\tan(x)$.*

]

9. Operations required in all flavors

This clause defines the required library arithmetic and non-arithmetic operations, see §6.2, of the standard. An implementation shall provide each required operation in each provided flavor.

For each arithmetic operation, this clause gives the mathematical formula for the point function, its domain of definition, and the set where it is continuous if different from the domain. The common evaluations of these operations are defined by this information according to case (a) in §7.3, and are not described for each individual operation.

For each non-arithmetic operation, the common evaluations are described explicitly.

The behavior of each operation outside the set of common evaluations is flavor-defined, subject to the general flavor requirements in §7.4, 7.5.

9.1. Arithmetic operations.

Table 9.1 on page 23 lists required arithmetic operations, including those normally written in function notation $f(x, y, \ldots)$ and 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.*]

Notes to Table 9.1

TABLE 9.1. Required forward elementary functions.
Each one is continuous at each point of its domain, except where stated in the Notes. Square and round brackets are used to include or exclude an interval bound, e.g., $(-\pi, \pi]$ denotes $\{ x \in \mathbb{R} \mid -\pi < x \le \pi \}$.

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

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 of one real variable x, 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 has no value. It is continuous at each point of its domain, including the positive y axis which is on the boundary of the domain.
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 outside the mathematical range.
f. $\operatorname{atan2}(y, x)$ is the principal value of the argument (polar angle) of (x, y) in the plane. It is discontinuous on the half-line $y = 0, x < 0$ contained within its domain.
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. $\texttt{sign}(x)$ is -1 if $x < 0$; 0 if $x = 0$; and 1 if $x > 0$. It is discontinuous at 0 in its domain.
i. $\texttt{ceil}(x)$ is the smallest integer $\geq x$. $\texttt{floor}(x)$ is the largest integer $\leq x$. $\texttt{trunc}(x)$ is the nearest integer to x in the direction of zero. `ceil` and `floor` are discontinuous at each integer. `trunc` is discontinuous at each nonzero integer. (As defined in the C standard §7.12.9.)
j. $\texttt{roundTiesToEven}(x), \texttt{roundTiesToAway}(x)$ are the nearest integer to x, with ties rounded to the even integer or away from zero, respectively. They are discontinuous at each $x = n + \frac{1}{2}$ where n is an integer. (As defined in the C standard §7.12.9.)
k. Smallest, or largest, of its real arguments.

9.2. Cancellative addition and subtraction.

For common intervals $\boldsymbol{x} = [\underline{x}, \overline{x}]$, $\boldsymbol{y} = [\underline{y}, \overline{y}]$, the operation $\texttt{cancelMinus}(\boldsymbol{x}, \boldsymbol{y})$ is defined if and only if the width of $\boldsymbol{x}$ is not less than that of $\boldsymbol{y}$, i.e., $\overline{x} - \underline{x} \geq \overline{y} - \underline{y}$, and is then the unique interval $\boldsymbol{z}$ such that $\boldsymbol{y} + \boldsymbol{z} = \boldsymbol{x}$, with formula $\boldsymbol{z} = [\underline{x} - \underline{y}, \overline{x} - \overline{y}]$. The operation $\texttt{cancelPlus}(\boldsymbol{x}, \boldsymbol{y})$ is equivalent to $\texttt{cancelMinus}(\boldsymbol{x}, -\boldsymbol{y})$.

9.3. Set operations.

For common intervals $\boldsymbol{x} = [\underline{x}, \overline{x}]$, $\boldsymbol{y} = [\underline{y}, \overline{y}]$:

– $\texttt{intersection}(\boldsymbol{x}, \boldsymbol{y})$ is the intersection $\boldsymbol{x} \cap \boldsymbol{y}$ if this is nonempty, with formula $[\max(\underline{x}, \underline{y}), \min(\overline{x}, \overline{y})]$. If the intersection is empty, no common value is defined.
– $\texttt{convexHull}(\boldsymbol{x}, \boldsymbol{y})$ is the tightest interval containing $\boldsymbol{x}$ and $\boldsymbol{y}$, with formula $[\min(\underline{x}, \underline{y}), \max(\overline{x}, \overline{y})]$.

9.4. Constructors. An interval constructor by definition is an operation that creates a bare or decorated interval from non-interval data.

A flavor shall provide means to construct common intervals as follows.

– The operation $\texttt{numsToInterval}(l, u)$, takes real values l and u. If the condition $l \leq u$ holds, its value is the common bare interval $[l, u] = \{\, x \in \mathbb{R} \mid l \leq x \leq u \,\}$. Otherwise, it has no common value.
– The operation $\texttt{textToInterval}(s)$ takes a text string s. If s is a valid interval literal with decorated value $\boldsymbol{x}_{\texttt{com}}$ in the set-based flavor, see §10.5.1, 12.11, its value is the common bare interval $\boldsymbol{x}$. Otherwise, it has no common value.

Each bare interval constructor shall have a corresponding decorated constructor that provides a decoration appropriate to the flavor. If the bare interval constructor has a bare common value $\boldsymbol{x}$, the decorated constructor has a decorated common value $\boldsymbol{x}_{\texttt{com}}$. Otherwise, it has no common value.

9.5. Numeric functions of intervals.

The operations in Table 9.2 are defined for all common intervals, with the formula shown.

9.6. Boolean functions of intervals.

The comparison relations in Table 9.3 shall be provided, whose value is a boolean (1 = true, 0 = false) result.

9.7. Operations on/with decorations. The function `newDec` initializes a bare interval:

$$\texttt{newDec}(\boldsymbol{x}) = \boldsymbol{x}_d$$

where d is a decoration, dependent on $\boldsymbol{x}$ and the flavor, such that the result is suitable input for decorated-interval evaluation of an expression in that flavor.

For a decorated interval $\boldsymbol{x}_{dx}$, the operations $\texttt{intervalPart}(\boldsymbol{x}_{dx})$ and $\texttt{decorationPart}(\boldsymbol{x}_{dx})$ shall be provided, with value $\boldsymbol{x}$ and dx, respectively.

TABLE 9.2. Required numeric functions of intervals.

Name	Definition
`inf`($\boldsymbol{x}$)	$\underline{x}$
`sup`($\boldsymbol{x}$)	$\overline{x}$
`mid`($\boldsymbol{x}$)	$(\underline{x} + \overline{x})/2$
`wid`($\boldsymbol{x}$)	$\overline{x} - \underline{x}$
`rad`($\boldsymbol{x}$)	$(\overline{x} - \underline{x})/2$
`mag`($\boldsymbol{x}$)	$\sup\{\, \lvert x \rvert \mid x \in \boldsymbol{x} \,\} = \max(\lvert\underline{x}\rvert, \lvert\overline{x}\rvert)$
`mig`($\boldsymbol{x}$)	$\inf\{\, \lvert x \rvert \mid x \in \boldsymbol{x} \,\} = \begin{cases} \min(\lvert\underline{x}\rvert, \lvert\overline{x}\rvert) & \text{if } \underline{x}, \overline{x} \text{ have the same sign} \\ 0 & \text{otherwise} \end{cases}$

TABLE 9.3. Comparisons for intervals $\boldsymbol{a}$ and $\boldsymbol{b}$. Notation $\forall_a$ means "for all a in $\boldsymbol{a}$", and so on. Column 4 gives formulae when $\boldsymbol{a}$=$[\underline{a}, \overline{a}]$ and $\boldsymbol{b}$=$[\underline{b}, \overline{b}]$ are common.

Name	Symbol	Defining predicate	Common $\boldsymbol{a}, \boldsymbol{b}$	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 $\boldsymbol{b}$
`subset`($\boldsymbol{a}, \boldsymbol{b}$)	$\boldsymbol{a} \subseteq \boldsymbol{b}$	$\forall_a \exists_b\, a = b$	$\underline{b} \le \underline{a} \wedge \overline{a} \le \overline{b}$	$\boldsymbol{a}$ is a subset of $\boldsymbol{b}$
`interior`($\boldsymbol{a}, \boldsymbol{b}$)	$\boldsymbol{a} \Subset \boldsymbol{b}$	$\forall_a \exists_b\, a < b \wedge \forall_a \exists_b\, b < a$	$\underline{b} < \underline{a} \wedge \overline{a} < \overline{b}$	$\boldsymbol{a}$ is interior to $\boldsymbol{b}$
`disjoint`($\boldsymbol{a}, \boldsymbol{b}$)	$\boldsymbol{a} \mathbin{\not\cap} \boldsymbol{b}$	$\forall_a \forall_b\, a \ne b$	$\overline{a} < \underline{b} \vee \overline{b} < \underline{a}$	$\boldsymbol{a}$ and $\boldsymbol{b}$ are disjoint

CHAPTER 2

Set-Based Intervals

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

10. Level 1 description

In this clause, subclauses §10.1 to §10.4 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 §10.5, 10.6 list the required and recommended *arithmetic operations* with their mathematical specifications.

10.1. Non-interval Level 1 entities. In addition to intervals, the required operations of this flavor handle entities of the following kinds, as inputs or outputs.

- 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.
- The **boolean** values `false`, `true`.
- The set of **decorations** defined in §11.

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

$$[\underline{x}, \overline{x}] = \{\, x \in \mathbb{R} \mid \underline{x} \le x \le \overline{x} \,\}, \tag{4}$$

where $\underline{x} = \inf \boldsymbol{x}$ and $\overline{x} = \sup \boldsymbol{x}$ are extended-real numbers satisfying $\underline{x} \le \overline{x}$, $\underline{x} < +\infty$ and $\overline{x} > -\infty$.

One calls $\underline{x}$ the **lower bound** and $\overline{x}$ the **upper bound** of the interval. Together they are its **bounds**. Conventionally $\emptyset$ has bounds $\inf \emptyset = +\infty$, $\sup \emptyset = -\infty$.

[*Notes.*

- *The definition implies $-\infty$ and $+\infty$ can be bounds of an interval, but are never members of it. In particular, (4) defines $[-\infty, +\infty]$ to be 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 a bound is excluded from an interval, e.g., $(a, b]$ or $]a, b]$ to denote $\{\, x \mid a < x \le b \,\}$. Where it is convenient to use this notation, it is pointed out, e.g., in the tables of function domains and ranges in §10.5, 10.6.*
- *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 (4) was preferred as it makes a one-to-one mapping between valid pairs $\underline{x}, \overline{x}$ of bounds and the nonempty intervals they specify.*

]

A **box** or **interval vector** is an n-tuple $(\boldsymbol{x}_1,\dots,\boldsymbol{x}_n)$ 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 $\mathbb{R}^n$; however, the correspondence is one-to-one only when all the $\boldsymbol{x}_j$ are nonempty.

In particular $x\in\boldsymbol{x}$, for $x\in\mathbb{R}^n$, means by definition $x_i\in\boldsymbol{x}_i$ for all $i=1,\dots,n$; and $\boldsymbol{x}$ is empty if and only if any of its components $\boldsymbol{x}_i$ is empty.

10.3. Hull. The (interval) **hull** of an arbitrary subset $\boldsymbol{s}$ of $\mathbb{R}^n$, written $\mathrm{hull}(\boldsymbol{s})$, 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.)

10.4. Functions and expressions. The terms *function*, *domain*, *range*, *point function*, *supported*, *operation*, *provided*, *arithmetic operation* have the meanings defined in §6.1, and *expression* has the meaning in §6.2.

Subclause 7.4 requires each flavor to define the Level 1 bare interval version and decorated interval version of any point function. In this flavor, the former is the natural interval extension defined here.

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

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

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

$$\boldsymbol{f}(\boldsymbol{x})=\mathrm{hull}(\mathrm{Rge}(f\,|\,\boldsymbol{x})).$$

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

$$\boldsymbol{f}(\boldsymbol{x}_1,\dots,\boldsymbol{x}_n)\supseteq\mathrm{Rge}(f\,|\,\boldsymbol{x}_1,\dots,\boldsymbol{x}_n),$$

and the natural interval extension is defined by

$$\boldsymbol{f}(\boldsymbol{x}_1,\dots,\boldsymbol{x}_n)=\mathrm{hull}(\mathrm{Rge}(f\,|\,\boldsymbol{x}_1,\dots,\boldsymbol{x}_n))$$

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

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. In this case, we refer to each of the above extensions as a *full Level 1 extension* of $\boldsymbol{f}$.

The natural extension is automatically defined for all interval or set arguments. The decoration system, Clause 11, 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

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

[*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.

From the above definition, an interval extension of a **real constant**—a zero-argument point function returning a real value c—is any zero-argument interval function that returns an interval containing c. Its *natural extension* returns the single-point interval $[c,c]$.

10.5. Required operations.

The operations listed in this subclause include those required in all flavors, see Clause 9. An implementation shall provide interval versions of them appropriate to its supported interval types. For constants and the forward and reverse arithmetic operations in §10.5.2, 10.5.3, 10.5.5, each such version shall be an interval extension (§10.4) 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 §10.5.9, is detailed in §12.9, 12.12.

The names of operations, as well as symbols used for operations (e.g., for the comparisons in §10.5.10), might not correspond to those that any particular language would use.

10.5.1. *Interval literals.*

An **interval literal** is a text string that denotes an interval. Level 1, which is mainly for human communication, merely assumes there exist some agreed rules on the form and meaning of interval literals. A specified form and meaning shall be used in Level 2 onward: the definition is in §12.11. This definition is also used in Level 1 of this document for examples, where relevant. [*Example. This includes the inf-sup form* `[1.234e5,Inf]`*; the uncertain form* `3.1416?1`*; and the named interval constant* `[Empty]`.]

10.5.2. *Interval constants.*

The constant functions `empty`() and `entire`() have value Empty and Entire, respectively.

10.5.3. *Forward-mode elementary functions.*

Table 9.1 on page 23 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 $x \bullet y$.

10.5.4. *Two-output division.*

The result of interval division $\boldsymbol{x}/\boldsymbol{y}$ considered as a set, that is

$$\boldsymbol{x} /_{\text{set}} \boldsymbol{y} = \{\, x/y \mid x \in \boldsymbol{x},\ y \in \boldsymbol{y} \text{ and } y \neq 0 \,\},$$

can have zero, one or two nonempty disjoint connected components, which need not be closed. [*Examples. Use the classical notation where a square or round bracket means a closed or open bound, respectively. Then*

$$\begin{aligned}
[1,2] /_{\text{set}} [0,0] &= \emptyset, \\
[1,2] /_{\text{set}} [1,1] &= [1,2], \\
[1,1] /_{\text{set}} [1,+\infty) &= (0,1], \\
[1,2] /_{\text{set}} [-1,1] &= (-\infty,-1] \cup [1,+\infty), \\
[1,1] /_{\text{set}} \text{Entire} &= (-\infty,0) \cup (0,+\infty),
\end{aligned}$$

are cases with 0, 1, 1, 2 and 2 output components, respectively. The third and fifth cases have components that are not closed.]

In applications such as the Interval Newton Method, it is useful to return enclosures of these components separately rather than the result of normal division which is the (closed) convex hull of their union, namely $\boldsymbol{x}/\boldsymbol{y} = \text{hull}(\boldsymbol{x} /_{\text{set}} \boldsymbol{y})$. The value of the operation $\texttt{divToPair}(\boldsymbol{x}, \boldsymbol{y})$ is an ordered pair $(\boldsymbol{u}, \boldsymbol{v})$ of closed intervals, namely

$$\texttt{divToPair}(\boldsymbol{x},\boldsymbol{y}) = \begin{cases} (\emptyset,\emptyset) & \text{if } \boldsymbol{x} /_{\text{set}} \boldsymbol{y} \text{ is empty,} \\ (\overline{\boldsymbol{u}},\emptyset) & \text{if } \boldsymbol{x} /_{\text{set}} \boldsymbol{y} \text{ has one component } \boldsymbol{u}, \\ (\overline{\boldsymbol{u}},\overline{\boldsymbol{v}}) & \text{if } \boldsymbol{x} /_{\text{set}} \boldsymbol{y} \text{ has two components } \boldsymbol{u}, \boldsymbol{v}, \text{ ordered so that } \boldsymbol{u} < \boldsymbol{v}, \end{cases}$$

where $\overline{\boldsymbol{a}}$ denotes the topological closure of $\boldsymbol{a}$, which in this case is equivalent to hull($\boldsymbol{a}$). [*Note.* `divToPair` *is not regarded as an arithmetic operation, since if it appears in an arithmetic expression, containment might be lost unless its two outputs are handled with care.*]

10.5.5. *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

$$\varphi\text{Rev}(\boldsymbol{c}, \boldsymbol{x}) \supseteq \{\, x \in \boldsymbol{x} \mid \varphi(x) \text{ is defined and in } \boldsymbol{c} \,\}, \tag{5}$$

TABLE 10.1. 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`$(\boldsymbol{c}, \boldsymbol{x})$	

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

$$\bullet\mathrm{Rev}_1(\boldsymbol{b}, \boldsymbol{c}, \boldsymbol{x}) \supseteq \{\, x \in \boldsymbol{x} \mid b \in \boldsymbol{b} \text{ exists such that } x \bullet b \text{ is defined and in } \boldsymbol{c} \,\}, \tag{6}$$

$$\bullet\mathrm{Rev}_2(\boldsymbol{a}, \boldsymbol{c}, \boldsymbol{x}) \supseteq \{\, x \in \boldsymbol{x} \mid a \in \boldsymbol{a} \text{ exists such that } a \bullet x \text{ is defined and in } \boldsymbol{c} \,\}. \tag{7}$$

If $\bullet$ is commutative, then $\bullet\mathrm{Rev}_1$ and $\bullet\mathrm{Rev}_2$ agree and may be implemented simply as $\bullet\mathrm{Rev}$.

In each of (5, 6, 7), 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 (5, 6, 7) 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 10.1. Note `pownRev`(x, p) is regarded as a family of unary functions parametrized by p.

[*Example.*

– *Consider the function $sqr(x) = x^2$. Evaluating $sqr\mathrm{Rev}([1, 4])$ answers the question: given that $1 \le x^2 \le 4$, 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*

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

– *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.

]

10.5.6. *Cancellative addition and subtraction.*

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 $\boldsymbol{s}_k = \sum_{i=1,\, i \ne k}^{n} \boldsymbol{a}_i$, 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 might incur many. Thus the assumption underlying these cancellative operations is that $\boldsymbol{x}$ is an enclosure of an unknown true sum $\boldsymbol{x}_0$, whereas $\boldsymbol{y}$ is "exact". The computed $\boldsymbol{z}$ is an enclosure of an unknown true $\boldsymbol{z}_0$ such that $\boldsymbol{y} + \boldsymbol{z}_0 = \boldsymbol{x}_0$.]

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

For any two bounded intervals $\boldsymbol{x}$ and $\boldsymbol{y}$, the value of the operation $\texttt{cancelMinus}(\boldsymbol{x}, \boldsymbol{y})$ is the tightest interval $\boldsymbol{z}$ such that

$$\boldsymbol{y} + \boldsymbol{z} \supseteq \boldsymbol{x}, \tag{8}$$

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

This specification leads to the following Level 1 algorithm. If $\boldsymbol{x} = \emptyset$ and $\boldsymbol{y}$ is bounded, 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 $\texttt{width}(\boldsymbol{x}) \geq \texttt{width}(\boldsymbol{y})$), 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 §B2.*]

10.5.7. *Set operations.*

The value of the operation $\texttt{intersection}(\boldsymbol{x}, \boldsymbol{y})$ is the intersection $\boldsymbol{x} \cap \boldsymbol{y}$ of the intervals $\boldsymbol{x}$ and $\boldsymbol{y}$.

The value of the operation $\texttt{convexHull}(\boldsymbol{x}, \boldsymbol{y})$ is the interval hull of the union $\boldsymbol{x} \cup \boldsymbol{y}$ of the intervals $\boldsymbol{x}$ and $\boldsymbol{y}$.

10.5.8. *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 $\texttt{numsToInterval}(l, u)$, takes extended-real values l and u. If (see §10.2) the conditions $l \leq u$, $l < +\infty$ and $u > -\infty$ hold, its value is the nonempty interval $[l, u] = \{ x \in \mathbb{R} \mid l \leq x \leq u \}$. Otherwise, it has no value.

The operation $\texttt{textToInterval}(s)$ takes a text string s. If s is a valid interval literal, see §10.5.1, 12.11, its value is the interval denoted by s. Otherwise, it has no value.

10.5.9. *Numeric functions of intervals.*

The operations in Table 10.2 shall be provided, the argument being an interval and the result a number, which for some of the operations might be infinite.
[*Note. Implementations should provide an operation that returns* $\texttt{mid}(\boldsymbol{x})$ *and* $\texttt{rad}(\boldsymbol{x})$ *simultaneously.*]

TABLE 10.2. Required numeric functions of an interval $\boldsymbol{x} = [\underline{x}, \overline{x}]$.
Note $\texttt{sup}$ can have value $-\infty$; each of $\texttt{inf}$, $\texttt{wid}$, $\texttt{rad}$ and $\texttt{mag}$ can have value $+\infty$.

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

10.5.10. *Boolean functions of intervals.*

The following operations shall be provided, whose value is a boolean (1 = true, 0 = false) result.

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

There are eight boolean-valued comparison relations, which take two interval inputs. These are defined in Table 10.3, in which column three gives the set-theoretic definition, and column four gives an equivalent specification when both intervals are nonempty. Table 10.4 shows what the definitions imply when at least one interval is empty.

TABLE 10.3. 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	Defining predicate	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 $\boldsymbol{b}$
`subset`$(\boldsymbol{a},\boldsymbol{b})$	$\boldsymbol{a} \subseteq \boldsymbol{b}$	$\forall_a \exists_b\, a = b$	$\underline{b} \le \underline{a} \wedge \overline{a} \le \overline{b}$	$\boldsymbol{a}$ is a subset of $\boldsymbol{b}$
`less`$(\boldsymbol{a},\boldsymbol{b})$	$\boldsymbol{a} \le \boldsymbol{b}$	$\forall_a \exists_b\, a \le b \wedge \forall_b \exists_a\, a \le b$	$\underline{a} \le \underline{b} \wedge \overline{a} \le \overline{b}$	$\boldsymbol{a}$ is weakly less than $\boldsymbol{b}$
`precedes`$(\boldsymbol{a},\boldsymbol{b})$	$\boldsymbol{a} \preccurlyeq \boldsymbol{b}$	$\forall_a \forall_b\, a \le b$	$\overline{a} \le \underline{b}$	$\boldsymbol{a}$ is to left of but may touch $\boldsymbol{b}$
`interior`$(\boldsymbol{a},\boldsymbol{b})$	$\boldsymbol{a} \Subset \boldsymbol{b}$	$\forall_a \exists_b\, a < b \wedge \forall_a \exists_b\, b < a$	$\underline{b} <' \underline{a} \wedge \overline{a} <' \overline{b}$	$\boldsymbol{a}$ is interior to $\boldsymbol{b}$
`strictLess`$(\boldsymbol{a},\boldsymbol{b})$	$\boldsymbol{a} < \boldsymbol{b}$	$\forall_a \exists_b\, a < b \wedge \forall_b \exists_a\, a < b$	$\underline{a} <' \underline{b} \wedge \overline{a} <' \overline{b}$	$\boldsymbol{a}$ is strictly less than $\boldsymbol{b}$
`strictPrecedes`$(\boldsymbol{a},\boldsymbol{b})$	$\boldsymbol{a} \prec \boldsymbol{b}$	$\forall_a \forall_b\, a < b$	$\overline{a} < \underline{b}$	$\boldsymbol{a}$ is strictly to left of $\boldsymbol{b}$
`disjoint`$(\boldsymbol{a},\boldsymbol{b})$	$\boldsymbol{a} \not\cap \boldsymbol{b}$	$\forall_a \forall_b\, a \neq b$	$\overline{a} < \underline{b} \vee \overline{b} < \underline{a}$	$\boldsymbol{a}$ and $\boldsymbol{b}$ are disjoint

TABLE 10.4. Comparisons with empty intervals.

	$\boldsymbol{a} = \emptyset$ $\boldsymbol{b} \neq \emptyset$	$\boldsymbol{a} \neq \emptyset$ $\boldsymbol{b} = \emptyset$	$\boldsymbol{a} = \emptyset$ $\boldsymbol{b} = \emptyset$
$\boldsymbol{a} = \boldsymbol{b}$	0	0	1
$\boldsymbol{a} \subseteq \boldsymbol{b}$	1	0	1
$\boldsymbol{a} \le \boldsymbol{b}$	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
$\boldsymbol{a} \not\cap \boldsymbol{b}$	1	1	1

[*Notes.*
- *Column two of Table 10.3 gives suggested symbols for use in typeset algorithms.*
- *All these relations, except $\boldsymbol{a} \not\cap \boldsymbol{b}$, are transitive for* nonempty *intervals.*
- *The first three are reflexive.*
- `interior` *uses the topological definition: $\boldsymbol{b}$ is a neighbourhood of each point of $\boldsymbol{a}$. This implies, for instance, that* `interior`*(Entire,Entire) is true.*
- *All occurrences of $<$ in column 4 of Table 10.3 can be replaced by $<'$.*

]

10.6. Recommended operations.

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.

10.6.1. *Forward-mode elementary functions.*

The list of recommended functions is in Table 10.5. Each interval version shall be an interval extension of the point function.

TABLE 10.5. Recommended elementary functions.
Normal mathematical notation is used to include or exclude an interval bound, 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 $\sqrt[q]{x}$, $q \in \mathbb{Z} \setminus \{0\}$	$\begin{cases} \mathbb{R} \text{ if } q > 0 \text{ odd} \\ [0, \infty) \text{ if } q > 0 \text{ even} \\ \mathbb{R}\setminus\{0\} \text{ if } q < 0 \text{ odd} \\ (0, \infty) \text{ if } q < 0 \text{ even} \end{cases}$	same as domain	a
$\begin{cases} \texttt{expm1}(x) \\ \texttt{exp2m1}(x) \\ \texttt{exp10m1}(x) \end{cases}$	b^x-1	$\mathbb{R}$	$(-1, \infty)$	b, c
$\begin{cases} \texttt{logp1}(x) \\ \texttt{log2p1}(x) \\ \texttt{log10p1}(x) \end{cases}$	$\log_b(x+1)$	$(-1, \infty)$	$\mathbb{R}$	b, c
`compoundm1`(x, y)	$(1 + x)^y - 1$	$\{x>-1\} \cup \{x=-1, y>0\}$	$[-1, \infty)$	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)	$\tan(\pi x)$	$\mathbb{R}\setminus\{ k + \frac{1}{2} \mid k \in \mathbb{Z} \}$	$\mathbb{R}$	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 10.5

a. Regarded as a family of functions of one real variable x, parameterized by the integer argument q.
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 9.1, note e.
f. To avoid confusion with notation for open intervals, in this table coordinates in $\mathbb{R}^2$ are delimited by angle brackets $\langle\ \rangle$.

10.6.2. *Slope functions.*

The functions in Table 10.6 are the commonest ones needed for efficient implementation of 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.

10.6.3. *Boolean functions of intervals.*

The following operations should be provided, whose value is a boolean (1 = true, 0 = false) result.

There is a function `isCommonInterval`$(\boldsymbol{x})$, with value 1 if $\boldsymbol{x}$ is a common interval (§7.2) and 0 otherwise. There is a function `isSingleton`$(\boldsymbol{x})$, with value 1 if $\boldsymbol{x}$ is the set of a single real and 0 otherwise.

There is a function `isMember`$(m, \boldsymbol{x})$, where m is an extended real and $\boldsymbol{x}$ is an interval. Its value is 1 if m is a finite real and m is a member of the set $\boldsymbol{x}$, 0 otherwise.

Table 10.6. Recommended slope functions.

Name	Definition	Point function domain	Point function range
`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)	$\frac{3}{x^3}(\sinh x - x)$	$\mathbb{R}$	$[\frac{1}{2}, \infty)$

10.6.4. *Extended interval comparisons.*

The **interval overlapping function** `overlap`$(\boldsymbol{a}, \boldsymbol{b})$, also written $\boldsymbol{a} \circledcirc \boldsymbol{b}$, 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 bounds move continuously with strictly positive velocity. Then, depending on the relative sizes of $\boldsymbol{a}$ and $\boldsymbol{b}$, the value of $\boldsymbol{a} \circledcirc \boldsymbol{b}$ 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 $\underline{a} < \underline{b} < \overline{a} < \overline{b}$.

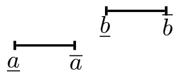
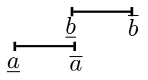
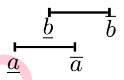
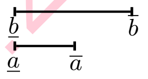
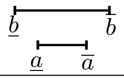
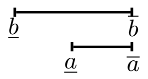
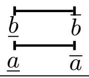
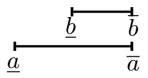
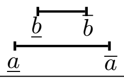
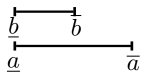
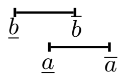
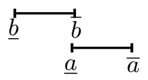
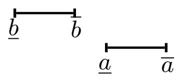
The three extra values are: `bothEmpty` when $\boldsymbol{a} = \boldsymbol{b} = \emptyset$, else `firstEmpty` when $\boldsymbol{a} = \emptyset$, `secondEmpty` when $\boldsymbol{b} = \emptyset$.

Table 10.7 shows the 16 states, with the 13 "nonempty" states specified (a) in terms of set membership using quantifiers and (b) in terms of the bounds $\underline{a}, \overline{a}, \underline{b}, \overline{b}$, and also (c) shown diagrammatically.

The "set" and "bound" columns in Table 10.7 remove some ambiguities of the diagram view when one interval shrinks to a single point that coincides with a bound of the other. Such a case is allocated to `equal` when all four bounds 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.*

Table 10.7. 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	Bound 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`	$\forall_a \forall_b\ a < b$	$\overline{a} < \underline{b}$	
`meets`	$\forall_a \forall_b\ a \leq b$ $\exists_a \forall_b\ a < b$ $\exists_a \exists_b\ a = b$	$\underline{a} < \overline{a}$ $\overline{a} = \underline{b}$ $\underline{b} < \overline{b}$	
`overlaps`	$\exists_a \forall_b\ a < b$ $\exists_b \forall_a\ a < b$ $\exists_a \exists_b\ b < a$	$\underline{a} < \underline{b}$ $\underline{b} < \overline{a}$ $\overline{a} < \overline{b}$	
`starts`	$\exists_a \forall_b\ a \leq b$ $\exists_b \forall_a\ b \leq a$ $\exists_b \forall_a\ a < b$	$\underline{a} = \underline{b}$ $\overline{a} < \overline{b}$	
`containedBy`	$\exists_b \forall_a\ b < a$ $\exists_b \forall_a\ a < b$	$\underline{b} < \underline{a}$ $\overline{a} < \overline{b}$	
`finishes`	$\exists_b \forall_a\ b < a$ $\exists_a \forall_b\ b \leq a$ $\exists_b \forall_a\ a \leq b$	$\underline{b} < \underline{a}$ $\overline{a} = \overline{b}$	
`equal`	$\forall_a \exists_b\ a = b$ $\forall_b \exists_a\ b = a$	$\underline{a} = \underline{b}$ $\overline{a} = \overline{b}$	
`finishedBy`	$\exists_a \forall_b\ a < b$ $\exists_b \forall_a\ a \leq b$ $\exists_a \forall_b\ b \leq a$	$\underline{a} < \underline{b}$ $\overline{b} = \overline{a}$	
`contains`	$\exists_a \forall_b\ a < b$ $\exists_a \forall_b\ b < a$	$\underline{a} < \underline{b}$ $\overline{b} < \overline{a}$	
`startedBy`	$\exists_b \forall_a\ b \leq a$ $\exists_a \forall_b\ a \leq b$ $\exists_a \forall_b\ b < a$	$\underline{b} = \underline{a}$ $\overline{b} < \overline{a}$	
`overlappedBy`	$\exists_b \forall_a\ b < a$ $\exists_a \forall_b\ b < a$ $\exists_b \exists_a\ a < b$	$\underline{b} < \underline{a}$ $\underline{a} < \overline{b}$ $\overline{b} < \overline{a}$	
`metBy`	$\forall_b \forall_a\ b \leq a$ $\exists_b \exists_a\ b = a$ $\exists_b \forall_a\ b < a$	$\underline{b} < \overline{b}$ $\overline{b} = \underline{a}$ $\underline{a} < \overline{a}$	
`after`	$\forall_b \forall_a\ b < a$	$\overline{b} < \underline{a}$	

For instance, suppose we make the states s in Table 10.7'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* `disjoint`$(\boldsymbol{a}, \boldsymbol{b})$. *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 [**7**]. *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.*]

11. The decoration system at Level 1

11.1. 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 throughout Clause 12. Subclauses §11.2, 11.3, 11.4 give the basic concepts. §11.5, 11.6 define how intervals are given an initial decoration, and the binding of decorations to library interval arithmetic operations to give correct propagation through expressions. §11.7 is about non-arithmetic operations. §11.5 describes housekeeping operations on decorations, including comparisons, and conversion between a decorated interval and its interval and decoration parts. §11.8 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; §11.9 gives explanatory notes on the `com` decoration. §11.10 defines a restricted decorated arithmetic that suffices for some important applications and is easier to implement efficiently.

In Annex B, §B6 gives examples of the meaning and use of decorations; and §B8 contains a rigorous theoretical foundation, including a proof of the Fundamental Theorem of Interval Arithmetic for this flavor.

11.2. 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 $n\ge 0$ 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:

$$d=\{\,(f,\boldsymbol{x})\mid p_d(f,\boldsymbol{x})\text{ is true}\,\}. \qquad (9)$$

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

Value	Short description	Property	Definition
`com`	common	$p_{\mathtt{com}}(f,\boldsymbol{x})$	$\boldsymbol{x}$ 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	$p_{\mathtt{dac}}(f,x)$	$\boldsymbol{x}$ is a nonempty subset of $\mathrm{Dom}(f)$, and the restriction of f to $\boldsymbol{x}$ is continuous;
`def`	defined	$p_{\mathtt{def}}(f,\boldsymbol{x})$	$\boldsymbol{x}$ is a nonempty subset of $\mathrm{Dom}(f)$;
`trv`	trivial	$p_{\mathtt{trv}}(f,\boldsymbol{x})$	always true (so gives no information);
`ill`	ill-formed	$p_{\mathtt{ill}}(f,\boldsymbol{x})$	Not an Interval; formally $\mathrm{Dom}(f)=\emptyset$, see §11.3.

(10)

These are listed according to the propagation order (19), 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 $\big((\boldsymbol{u}_1,d_1),\dots,(\boldsymbol{u}_n,d_n)\big)$, 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 $\boldsymbol{u},\boldsymbol{v},\dots$ are often named $du,dv,\dots$ 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 (9), `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. (11)

Namely the definitions (10) give:

$$\texttt{com} \subset \texttt{dac} \subset \texttt{def} \subset \texttt{trv} \supset \texttt{ill}, \qquad \text{note the change from } \subset \text{ to } \supset; \tag{12}$$

$$\texttt{com}, \texttt{dac} \text{ and } \texttt{def} \text{ are disjoint from } \texttt{ill}. \tag{13}$$

Property (11) implies that for any $(f, \boldsymbol{x})$ there is a unique tightest (in the containment order (12)), 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:

$$\operatorname{Dec}(f \,|\, \boldsymbol{x}) = d \iff p_d(f, \boldsymbol{x}) \text{ holds, but } p_e(f, \boldsymbol{x}) \text{ fails for all } e \subset d. \tag{14}$$

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

11.3. 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 C3.

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 has no value at Level 1.

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 provide means for an exception to be signaled when an NaI is produced.

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

11.4. Permitted combinations. A decorated interval $\boldsymbol{y}_{dy}$ shall always be such that $\boldsymbol{y} \supseteq \operatorname{Rge}(f \,|\, \boldsymbol{x})$ and $p_{dy}(f, \boldsymbol{x})$ holds, for some $(f, \boldsymbol{x})$ as in §11.2—informally, it must tell the truth about some conceivable evaluation of a function over a box. If $dy = \texttt{def}$, `dac` or `com` then by definition $\boldsymbol{x}$ is nonempty, and f is everywhere defined on it, so that $\operatorname{Rge}(f \,|\, \boldsymbol{x})$ is nonempty, implying $\boldsymbol{y}$ is nonempty. Hence, these decorated intervals are contradictory: $\emptyset_{\texttt{dac}}$ and $\emptyset_{\texttt{def}}$; and $\boldsymbol{x}_{\texttt{com}}$ if $\boldsymbol{x}$ is empty or unbounded. Implementations shall not produce them.

No other combinations are essentially forbidden.

11.5. Operations on/with decorations. This subclause contains operations to initialize the decoration on a bare interval and to disassemble and reassemble a decorated interval; and comparisons for decorations.

11.5.1. *Initializing.* 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.

The simplest expression with one argument x is the trivial expression "x" with no operations. It defines the identity function Id that maps a real x to itself, $\operatorname{Id}(x) = x$. For interval-evaluation of this expression over some bare interval $\boldsymbol{x}$, the appropriate initial decoration for $\boldsymbol{x}$ is the strongest decoration d that makes $p_d(\operatorname{Id}, \boldsymbol{x})$ true, that is

$$d = \operatorname{Dec}(\operatorname{Id} \,|\, \boldsymbol{x}) = \begin{cases} \texttt{com} & \text{if } \boldsymbol{x} \text{ is nonempty and bounded,} \\ \texttt{dac} & \text{if } \boldsymbol{x} \text{ is unbounded,} \\ \texttt{trv} & \text{if } \boldsymbol{x} \text{ is empty.} \end{cases}$$

The function `newDec`() constructs a decorated interval from a bare one by adding such an initial decoration:

$$\texttt{newDec}(\boldsymbol{x}) = \boldsymbol{x}_d \quad \text{where} \quad d = \operatorname{Dec}(\operatorname{Id} \,|\, \boldsymbol{x}). \tag{15}$$

Initializing each input thus, before evaluating an expression, ensures the most informative decoration on the output.

11.5.2. *Disassembling and assembling.* For a decorated interval $\boldsymbol{x}_{dx}$, the operations `intervalPart`$(\boldsymbol{x}_{dx})$ and `decorationPart`$(\boldsymbol{x}_{dx})$ shall be provided, with value $\boldsymbol{x}$ and dx, respectively. For the case of NaI, `decorationPart`(NaI) has the value `ill`, but `intervalPart`(NaI) has no value at Level 1.

Given an interval $\boldsymbol{x}$ and a decoration dx, the operation `setDec`$(\boldsymbol{x}, dx)$ returns the decorated interval $\boldsymbol{x}_{dx}$ if this is an allowed combination. The cases of forbidden combinations are as follows:

– `setDec`$(\emptyset, dx)$ where dx is one of `def`, `dac` or `com` returns $\emptyset_{\texttt{trv}}$.
– `setDec`$(\boldsymbol{x}, \texttt{com})$, for any unbounded $\boldsymbol{x}$, returns $\boldsymbol{x}_{\texttt{dac}}$.
– `setDec`$(\boldsymbol{x}, \texttt{ill})$ for any $\boldsymbol{x}$, whether empty or not, returns NaI.

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

11.5.3. *Comparisons.* 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 (19).

11.6. 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

$$\boldsymbol{w} \supseteq \mathrm{Rge}(\varphi \,|\, \boldsymbol{v}), \qquad \text{(enclosure).} \tag{16}$$

The decorated interval extension of φ determines a dv_0 such that

$$p_{dv_0}(\varphi, \boldsymbol{v}) \text{ holds}, \qquad \text{(a "local decoration").} \tag{17}$$

It then evaluates the output decoration dw by

$$dw = \min\{dv_0, dv_1, \ldots, dv_k\}, \qquad \text{(the "min-rule"),} \tag{18}$$

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

$$\texttt{com} > \texttt{dac} > \texttt{def} > \texttt{trv} > \texttt{ill}. \tag{19}$$

[*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* $\mathsf{f}(z_1, \ldots, z_n)$ *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*

$$\boldsymbol{y} \supseteq \mathrm{Rge}(f \,|\, \boldsymbol{x}) \tag{20}$$

but also

$$p_{dy}(f, \boldsymbol{x}) \text{ holds.} \tag{21}$$

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 Interval Arithmetic** *(FTIA). The rules for initializing and propagating decorations are key to its validity. They are justified, and a formal statement and proof of the FTIA given, in Annex B.*

Briefly, (15) 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 (18) 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 (16) is compatible with many bare interval extensions, typically coming from different interval types at Level 2, so there might be several dv_0 satisfying the local decoration requirement (17). The ideal choice is the strongest decoration d such that $p_d(\varphi, \boldsymbol{v})$ holds, that is to take*

$$dv_0 = \mathsf{Dec}(\varphi \,|\, \boldsymbol{v}). \tag{22}$$

This is easily computable in finite precision for the arithmetic operations in §10.5, 10.6—see the tables in Annex B, §B5. However, functions may be added to the library in the future for which (22) is impractical to compute for some arguments $\boldsymbol{v}$. Hence, the weaker requirement (17) is made.

]

11.7. Decoration of non-arithmetic operations.

Interval-valued operations. These give interval results but are not interval extensions of point functions:

– the reverse-mode operations of §10.5.5;
– the cancellative operations `cancelPlus`$(\boldsymbol{x}, \boldsymbol{y})$ and `cancelMinus`$(\boldsymbol{x}, \boldsymbol{y})$ of §10.5.6;
– The set-oriented operations `intersection`$(\boldsymbol{x}, \boldsymbol{y})$ and `convexHull`$(\boldsymbol{x}, \boldsymbol{y})$ of §10.5.7.

No one way of decorating these operations gives useful information in all contexts. Therefore, a *trivial* decorated interval version is provided as follows. If any input is NaI, the result is NaI; otherwise the corresponding operation is applied to the interval parts of the inputs, and its result decorated with `trv`. The user may replace this by a nontrivial decoration via `setDec`(), see §11.5, where this can be deduced in a given application.

Non-interval-valued operations. These give non-interval results:

– the numeric functions of §10.5.9;
– the boolean-valued functions of §10.5.10;
– the overlap function of §10.6.4.

For each such operation, if any input is NaI, the result has no value at Level 1. Otherwise, the operation acts on decorated intervals by discarding the decoration and applying the corresponding bare interval operation.

11.8. User-supplied functions. A user may define a decorated interval extension of some point function, as defined in §11.6, 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 $\mathsf{Rge}(f \,|\, \boldsymbol{x}) = [2, 2\frac{1}{2}]$.

Thus it is useful to code a tight interval extension by special methods, e.g., monotonicity arguments, and to 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 (18). In this case, it is straightforward to compute the strongest local decoration $d = \mathsf{Dec}(f \,|\, \boldsymbol{x})$*, as follows.*

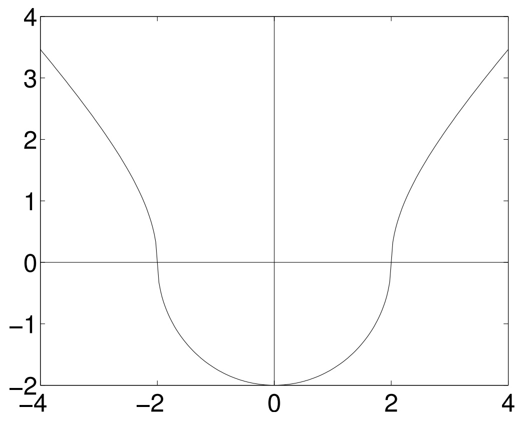
$$d = \begin{cases} \texttt{com} & \textit{if } 0 \notin \boldsymbol{x} \textit{ and } \boldsymbol{x} \textit{ is nonempty and bounded,} \\ \texttt{dac} & \textit{if } 0 \notin \boldsymbol{x} \textit{ and } \boldsymbol{x} \textit{ is unbounded,} \\ \texttt{trv} & \textit{if } 0 \in \boldsymbol{x} \textit{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

(ii)

The next example shows how an expert might 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 $\boldsymbol{y}_{dy} = f(\boldsymbol{x}_{dx})$
$\boldsymbol{u} = f_1(\boldsymbol{x} \cap [-\infty, -2])$
$\boldsymbol{v} = f_2(\boldsymbol{x} \cap [-2, 2])$
$\boldsymbol{w} = f_1(\boldsymbol{x} \cap [2, +\infty])$
$\boldsymbol{y} = \boldsymbol{u} \cup \boldsymbol{v} \cup \boldsymbol{w}$
$dy = dx$
```

Here $\cup$ *denotes the* `convexHull` *operation. 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.*

]

11.9. 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 might 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 might 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.

]

11.10. Compressed arithmetic with a threshold (optional).

11.10.1. *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. Which compressed interval types are provided is implementation-defined.

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 **threshold** τ, where $\tau \geq$ `trv` in the propagation order (19):

$dy \geq \tau$: 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 exceptional condition to have occurred. The interval part is not used, but one exploits the information given by the decoration.

11.10.2. *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 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 (10), 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;

(23)

`emp` lies between `trv` and `ill` in the containment order (12) and the propagation order (19):

$$\begin{gathered}\texttt{com} \subset \texttt{dac} \subset \texttt{def} \subset \texttt{trv} \supset \texttt{emp} \supset \texttt{ill}, \\ \texttt{com} > \texttt{dac} > \texttt{def} > \texttt{trv} > \texttt{emp} > \texttt{ill}.\end{gathered} \tag{24}$$

Since $\tau \geq$ `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 exceptional conditions 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.]

11.10.3. *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 $d = \texttt{ill}$, return $\text{Empty}_{\texttt{ill}} = \text{NaI}$
  elseif $d = \texttt{emp}$, return $\text{Empty}_{\texttt{trv}}$
  else return $\text{Entire}_d$
```

Arithmetic operations on compressed intervals shall follow *worst case semantics* rules that treat a decoration in $\{\texttt{trv}, \texttt{def}, \texttt{dac}\}$ 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 decoration dy, otherwise the bare interval $\boldsymbol{y}$.
– For operations with at least one decoration input, the result is always a decoration. A bare interval input is treated as in the previous item. A decoration d in $\{\texttt{emp}, \texttt{ill}\}$ is treated as $\emptyset_d$. A decoration d in $\{\texttt{trv}, \texttt{def}, \texttt{dac}\}$ 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 (24)) 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.

[*Example. Assuming* $\tau >$ `def`,
– *The division* `def`$/[1, 2]$ *becomes* $\boldsymbol{x}_{\texttt{def}}/[1, 2]_\tau$ *with arbitrary nonempty interval* $\boldsymbol{x}$.
The result is always decorated `def`, *so returns* `def`.
– *But* $[1, 2]/$`def` *becomes* $[1, 2]_\tau/\boldsymbol{x}_{\texttt{def}}$ *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. Sample tables and worked examples are in Annex B7.

If compressed arithmetic is implemented, it shall provide versions of all the required operations of §10.5, and it should provide the recommended operations of §10.6.

12. Level 2 description

12.1. Level 2 introduction. Entities and operations at Level 2 are said to have **finite precision**. From them, implementable interval algorithms may be constructed. Level 2 entities 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 decoration, string and boolean datums.

12.1.1. *Types and formats.* Following 754 terminology, numeric (usually but not necessarily 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. If $\mathbb{F}$ denotes a format, an **$\mathbb{F}$-number** means a member of $\mathbb{F}$, possibly infinite but not the "not a number" value NaN; to allow it to be NaN, it is called an **$\mathbb{F}$-datum**. If $\mathbb{T}$ denotes a bare or decorated interval type, a **$\mathbb{T}$-interval** means a member of $\mathbb{T}$, possibly empty but (in the decorated case) not the "not an interval" value NaI; to allow it to be NaI, it is called a **$\mathbb{T}$-datum**. A **$\mathbb{T}$-box** means a box with $\mathbb{T}$-datum components—equivalently, with $\mathbb{T}$-interval components, if $\mathbb{T}$ is a bare interval type.

The standard defines three kinds of interval type:

- **Bare interval types**, see §12.5, are named finite sets of (mathematical, Level 1) intervals.
- **Decorated interval types**, also see §12.5, are named finite sets of decorated intervals.
- **Compressed interval types** (optional) are named finite sets of compressed intervals. They are described in §11.10, and Level 2 makes no further requirements on them.

For each bare interval type there shall be a corresponding **derived** decorated interval type, and each decorated interval type shall be derived from a bare interval type, see §12.5.1. An implementation shall provide at least one supported bare interval type. If 754-conforming, it shall provide the inf-sup type, see §12.5.2, of at least one of the five basic formats of 754 §3.3. Beyond this, which types are provided is language- or implementation-defined.

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

12.1.2. *Operations.* A Level 2 operation is a finite-precision approximation to the corresponding Level 1 operation. To describe the required functionality, the standard treats each Level 1 operation as having a number of Level 2 **versions** in which each interval input or output is given a specific interval type, and each numeric input or output is given a specific numeric format.
[*Note. An implementation might provide the functionality via differently named operations for each version, or by overloading a single operation name, or by a single operation that accepts all the required type/format combinations, or in other ways.*

For example, let $\mathbb{T}_1$ and $\mathbb{T}_2$ be two supported interval types. The standard requires a version of addition $\boldsymbol{z} = \boldsymbol{x} + \boldsymbol{y}$ where each of $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}$ has type $\boldsymbol{T}_1$, and another where each of them has type $\boldsymbol{T}_2$. An implementation might provide this functionality by having two separate operations; another might have a single operation that takes inputs of types $\mathbb{T}_1$ and $\mathbb{T}_2$ in any combination, with some rule to determine the type of the output $\boldsymbol{z}$; etc.]

The term **$\mathbb{T}$-version** of a Level 1 operation denotes one in which any input or output that is an interval, is a $\mathbb{T}$-datum. For bare interval types this includes the following:

(a) A $\mathbb{T}$-interval extension (§12.9) of one of the required or recommended arithmetic operations of §10.5, 10.6.
(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 numeric.
(d) A constructor, whose input is numeric or text and output is a $\mathbb{T}$-datum.
(e) The operations of §13.3, 13.4, whose input is a $\mathbb{T}$-interval and output is a string.

Additionally, a version of `convertType` with output of type $\mathbb{T}$, see §12.12.10, is a $\mathbb{T}$-version irrespective of the type of the input interval. Generically these comprise the **operations of the type** $\mathbb{T}$, for the implementation.

12.1.3. *Exception behavior.* For some operations, and some particular inputs, there might not be a valid result. At Level 1 there are several cases when no value exists. However, a Level 2

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

operation always returns a value. When the Level 1 result does not exist, the operation returns either

- a special value indicating this event (e.g., NaN for most of the numeric functions in §12.12.8); or
- a value considered reasonable in practice. For example, `mid`(Entire) returns 0; a constructor given invalid input returns Empty; and one of the comparisons of §12.12.9, if any input is NaI, returns `false`.

The standard defines the following **exceptions** that may be signaled, causing a handler to be invoked:

- For the interval constructors of §12.12.7, and for `exactToInterval` in §13.4, it is possible that at Level 2, using finite precision, the implementation cannot decide whether a Level 1 value exists or not, see *Difficulties in implementation* in §12.12.7. If the constructor is certain that no Level 1 value exists, the exception `UndefinedOperation` is signaled; if it cannot decide, the exception `PossiblyUndefinedOperation` is signaled.
- If `intervalPart`() is called with NaI as input, the exception `IntvlPartOfNaI` is signaled, see §12.12.11.
- If some inputs of an operation are invalid, the exception `InvalidOperand` might be signaled, see §14.3.

These exceptions are separate from the exceptional conditions handled by the decoration system. The mechanism of how a handler deals with an exception is language- or implementation-defined. An implementation may provide exception handling additional to that above.

12.2. Naming conventions for operations. An operation is generally given a name that suits the context. For example, the addition of two interval datums $\boldsymbol{x}, \boldsymbol{y}$ might be written in generic algebra notation $\boldsymbol{x} + \boldsymbol{y}$; or with a generic text name `add`$(\boldsymbol{x}, \boldsymbol{y})$; or giving full type information such as *infsup-decimal64*-`add`$(\boldsymbol{x}, \boldsymbol{y})$. It might also be written as $\mathbb{T}$-`add`$(\boldsymbol{x}, \boldsymbol{y})$ to show it is an operation of a particular but unspecified type $\mathbb{T}$.

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

12.3. 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 floating-point" for numbers—focusing on the Level 1 entities denoted, and hiding the Level 3 representation.

However, a datum is more than just the Level 1 value: for instance, the number 3.75 represented in 32 bit binary floating-point is a different datum from the same number represented in 64 bit binary floating-point ("single" and "double" precision respectively in typical implementations).

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

$$\text{number datum} = (\text{Level 1 number, format name}),$$
$$\text{bare interval datum} = (\text{Level 1 interval, type name}),$$

where the name is a symbol that uniquely identifies the format or type. Since a decorated interval combines a bare interval and a decoration it thus becomes a triple at Level 2:

$$\text{decorated interval datum} = (\text{Level 1 interval, type name, decoration}).$$

The Level 1 value is said to be **tagged** by the format or type 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.*]

The interval comparison operations of §10.5.10, including comparison for equality, are provided between datums $\boldsymbol{x}, \boldsymbol{y}$ of the same type. Additionally they are provided between datums of different types provided the types are *comparable*, see §12.5.1.

Therefore it is necessary to distinguish kinds of equality. $\boldsymbol{x}$ and $\boldsymbol{y}$ are **identical** datums if they have the same Level 1 value and the same type. If their types are comparable then `equal`$(\boldsymbol{x}, \boldsymbol{y})$ is defined for them; if they have the same Level 1 value, `equal`$(\boldsymbol{x}, \boldsymbol{y})$ returns `true` and they are

called equal. If their types are not comparable, $\texttt{equal}(\boldsymbol{x}, \boldsymbol{y})$ is undefined; they are not equal even if they have the same Level 1 value.
[*Note. This is like the situation for 754 floating-point numbers. For instance, the number* 3.75 *is representable exactly by datums* x, y, z *in* `binary32`, `binary64` *and* `decimal64`, *respectively. They are non-identical datums; but the comparison* $x = y$ *(equivalently* $\texttt{compareQuietEqual}(x, y)$*) is defined since* x *and* y *have the same radix, and returns* `true` *because they have the same Level 1 value. However,* $x = z$ *is not defined within the 754 standard, because* x *has a different radix from* z*.*

Similarly, let $\boldsymbol{x}$*,* $\boldsymbol{y}$ *and* $\boldsymbol{z}$ *be the datums in the inf-sup types of* `binary32`, `binary64` *and* `decimal64`, *respectively, for an interval that they all represent exactly, such as* $[1, 3.75]$*. They are non-identical datums; but the comparison* $\boldsymbol{x} = \boldsymbol{y}$ *(equivalently* $\texttt{equal}(\boldsymbol{x}, \boldsymbol{y})$*) is defined and returns* `true`*; while* $\boldsymbol{x} = \boldsymbol{z}$ *is not defined in this standard, though* $\boldsymbol{x}$ *and* $\boldsymbol{z}$ *have the same Level 1 value, because their types are not comparable.*]

For a decorated interval type $\mathbb{T}$, the unique NaI datum is equal to itself as a datum, but compares unequal to any $\mathbb{T}$-datum, including itself, with the `equal` relation. This follows the behavior of NaN among 754 floating-point datums.

12.4. Number formats. In view of §12.3, a **number format**, or just format, is formally the set of all pairs (x, f) such that x belongs to a given finite set $\mathbb{F}$ of numbers and symbols, and f is a name for the format. A format is **provided** if the implementation provides a representation of it as in §14.2, and is **supported** (by this standard) if in addition:

– $\mathbb{F}$ comprises a subset of the extended reals $\overline{\mathbb{R}}$ that includes $-\infty$ and $+\infty$, together with a value NaN, and optionally signed zeros -0 and $+0$.
– $\mathbb{F}$ contains 0, or contains -0 and $+0$, or both.
– $\mathbb{F}$ contains at least one nonzero finite number.
– $\mathbb{F}$ is symmetric: if x is in $\mathbb{F}$, so is its additive inverse $-x$, where ± 0 are additive inverses of each other, as are $\pm\infty$.

Following the convention of omitting names, the format is normally identified with the set $\mathbb{F}$, and one may say a supported format is a set comprising NaN together with a finite subset of $\overline{\mathbb{R}}$, and possibly ± 0, subject to the above rules. A member of $\mathbb{F}$ is called an **$\mathbb{F}$-datum**. Every member except NaN is called **numeric**, or an **$\mathbb{F}$-number**.

As an aid to notation, if x is an $\mathbb{F}$-number then $\mathrm{Val}(x)$, the (extended-real) **value** of x, is the member of $\overline{\mathbb{R}}$ that x denotes: $\mathrm{Val}(-0) = \mathrm{Val}(+0) = 0$, $\mathrm{Val}(x) = x$ for other $\mathbb{F}$-numbers, and $\mathrm{Val}(\mathrm{NaN})$ is undefined. Thus $\mathrm{Val}(\mathbb{F})$ is the whole set $\{\, \mathrm{Val}(x) \mid x \in \mathbb{F} \,\}$ of extended reals denoted by $\mathbb{F}$.

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 datums representable in that format. At Level 2, different kinds of NaN map to the unique NaN of the number format.

In this document the five basic formats of 754 §3.3 are named `binary32`, `binary64`, `binary128`, `decimal64`, `decimal128`. Abbreviated names such as `b64` instead of `binary64` are sometimes used, and refer to the same format.

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.

For a Level 2 operation, an input or output that in the corresponding Level 1 operation is an extended real becomes a datum of some format $\mathbb{F}$. On input, if this datum x is not NaN, it is replaced by $\mathrm{Val}(x)$ before applying the Level 1 operation; if NaN, it is handled by rules specific to the operation. On output, an extended-real Level 1 result is mapped (**rounded**) to an $\mathbb{F}$-datum according to rules specific to the operation (see §12.12.8).

12.5. Bare and decorated interval types.

12.5.1. *Definition.* In view of §12.3, a **bare interval type**, or just type, is formally the set of all pairs $(\boldsymbol{x}, t)$ such that $\boldsymbol{x}$ belongs to a given finite subset $\mathbb{T}$ of the mathematical intervals $\overline{\mathbb{IR}}$, and t is a name for the type. It is **provided** if the implementation provides a representation of it as in §14.2, and is **supported** (by this standard) if in addition:

– $\mathbb{T}$ contains Empty and Entire.
– $\mathbb{T}$ is symmetric: if $\boldsymbol{x}$ is in $\mathbb{T}$, so is $-\boldsymbol{x}$.

The **decorated interval type derived** from $\mathbb{T}$ is formally the set of triples $(\boldsymbol{x}, t, d)$ such that $(\boldsymbol{x}, t)$ is a $\mathbb{T}$-interval, and $d \in \mathbb{D}$ is a decoration that follows the rule for permitted combinations $(\boldsymbol{x}, d)$ in §11.4.

Following the convention of omitting names, the type is normally regarded as being the set $\mathbb{T}$, and one may say a bare interval type is an arbitrary set of intervals subject to the above rules. The derived decorated type is then regarded as a set of pairs $(\boldsymbol{x}, d)$, equivalently $\boldsymbol{x}_d$, where $\boldsymbol{x} \in \mathbb{T}$ and $d \in \mathbb{D}$.

Following 754's terminology for formats (754-2008 Definition 2.1.36), a type $\mathbb{T}'$ is **wider**[2] than a type $\mathbb{T}$ (and $\mathbb{T}$ is **narrower** than $\mathbb{T}'$) if $\mathbb{T}$ is a subset of $\mathbb{T}'$ when they are regarded as sets of Level 1 intervals, ignoring the type tags and possible decorations. Two types are **comparable** if either is wider than the other. [*Example. The basic 754-conforming types of a given radix are comparable, see §12.6.*]

Each decorated interval type shall contain a "Not an Interval" datum NaI, identified with $(\emptyset, \texttt{ill})$. It shall appear to be unique at Level 2, but non-Level-2 operations may be provided to set and get a payload in an NaI for diagnostic purposes, in an implementation-defined way (see §11.3).

[*Example. 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 usually would define these as different bare interval types, by tagging members of S_1 by one type name t_1 and members of S_2 by another name t_2—represented at Level 3 by a pair of `binary32` *or of* `binary64` *numbers, respectively. However, it might 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, say, for some large arrays. The resulting Level 1 intervals are exactly the same those of inf-sup* `binary64`*, so this is just another way to store the latter type; but an implementation would give it a different name, to reflect the different storage and hence different operations at the code level.*]

12.5.2. *Inf-sup and mid-rad types.*

The **inf-sup type** derived from a supported number format $\mathbb{F}$ (the type **inf-sup** $\mathbb{F}$, e.g., "inf-sup `binary64`") is the bare interval type $\mathbb{T}$ comprising all intervals whose bounds are in $\mathrm{Val}(\mathbb{F})$, together with Empty. When $\mathbb{F}$ is a 754 format, the **radix** of $\mathbb{T}$ means the radix of $\mathbb{F}$.

[*Note. This implies* Entire *is in $\mathbb{T}$ because $\pm\infty \in \mathbb{F}$ by the definition of a supported format, see §12.4, so $\mathbb{T}$ satisfies the requirements for a bare interval type given in §12.5.1.*]

A **mid-rad** bare interval type is one whose nonempty bounded intervals comprise all intervals of the form $[m - r, m + r]$, where $m \in \mathrm{Val}(\mathbb{F})$ for some number format $\mathbb{F}$, and $r \in \mathrm{Val}(\mathbb{F}')$ for a possibly different format $\mathbb{F}'$, with m, r finite and $r \geq 0$. From the definition in §12.5.1 such a type shall contain Empty and Entire (so at Level 3 it shall have representations of these). It may also contain semi-bounded intervals.

12.6. 754-conformance. The standard defines the notion of 754-conformance, whose stronger requirements improve accuracy and programming convenience.

12.6.1. *Definition.* A **754-conforming type** is an inf-sup type derived from a 754 floating-point format (one of the five basic formats or an extended precision or extendable precision format) in the sense of §12.5.2 that meets the general requirements for conformance and whose operations meet the accuracy requirements in §12.10.

A **754-conforming implementation** is one, all of whose types are 754-conforming, and whose operations meet the requirements for mixed-type arithmetic in the next paragraphs. It might be a conforming part of an implementation in the sense of §3.1.

12.6.2. *754-conforming mixed-type operations.* The 754 standard requires a conforming floating-point system to provide mixed-format *formatOf* operations, where 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.

A 754-conforming implementation shall provide corresponding mixed-type interval operations. Namely, if it provides types $\mathbb{T}, \mathbb{T}_1, \mathbb{T}_2, \ldots$ derived from 754 formats $\mathbb{F}, \mathbb{F}_1, \mathbb{F}_2, \ldots$ all of the same

[2]Wider means having more precision. In the 754 context, for a given radix, a wider format is one with a wider bit string for the exponent and/or significand in its Level 4 encoding.

radix, then for each *formatOf* operation with output format $\mathbb{F}$ and accepting input formats chosen from $\mathbb{F}_1, \mathbb{F}_2, \ldots$ there shall be an interval version of that operation with output type $\mathbb{T}$ and input types chosen from $\mathbb{T}_1, \mathbb{T}_2, \ldots$. The result shall be computed as if using the exact inputs and shall meet the accuracy requirement for that operation, specified in §12.10.
[*Note. For inf-sup types this requirement may be met by implicit widening of inputs to a common widest format, as double rounding is not an issue for directed rounding.*]

12.7. Multi-precision interval types. Multi-precision 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 bounds 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 §12.8.*]

A conforming implementation defines 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.

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

12.8.1. *Hull in one dimension.* For each bare interval type $\mathbb{T}$ there shall be defined an **interval hull** operation

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

also called the $\mathbb{T}$-hull, which is part of $\mathbb{T}$'s mathematical definition. For an implicit type, see below, the implementation's documentation shall specify the $\mathbb{T}$-hull, e.g., by an algorithm. For an explicit type, it is uniquely determined and need not be separately specified.
[*Note. An implementation provides* $\operatorname{hull}_{\mathbb{T}}$ *as the operation* `convertType` *for conversion between any two supported types, see §12.12.10.*]

The $\mathbb{T}$-hull 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

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

Since $\mathbb{T}$ is a finite set and contains Entire, such a minimal $\boldsymbol{y}$ exists for any $\boldsymbol{s}$. In general $\boldsymbol{y}$ might 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**.

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

The $\mathbb{T}$-hull operation **overflows** if it can only find an unbounded $\mathbb{T}$-interval $\boldsymbol{y}$ to enclose a bounded set $\boldsymbol{s}$. Similarly, any Level 2 interval-valued operation overflows when its Level 1 result is bounded but too large to be enclosed in a bounded interval of the output type.
[*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 (§12.5.2), 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 a function, it might be expensive.]

12.8.2. *Hull in several dimensions.* In n dimensions the $\mathbb{T}$-hull, as defined mathematically in §12.8.1, is extended to act componentwise, namely for an arbitrary subset $\boldsymbol{s}$ of $\mathbb{R}^n$ it is $\operatorname{hull}_{\mathbb{T}}(\boldsymbol{s}) = (\boldsymbol{y}_1, \ldots, \boldsymbol{y}_n)$ where

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

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 $\boldsymbol{s}$.

12.9. 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

$$\boldsymbol{f}(\boldsymbol{x}) \supseteq \operatorname{Rge}(f \mid \boldsymbol{x}) \tag{25}$$

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 (§10.4) so that this document does not distinguish between Level 2 and Level 1 extensions. Namely define $\boldsymbol{f}^*$ by

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

for any subset $\boldsymbol{s}$ of $\mathbb{R}^n$. Then the interval $\boldsymbol{f}^*(\boldsymbol{s})$ contains $\operatorname{Rge}(f \mid \boldsymbol{s})$ 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.

12.10. Accuracy of operations. This subclause describes requirements and recommendations on the accuracy of operations. Here, *operation* denotes any Level 2 version, provided by the implementation, of a Level 1 operation with interval output and at least one interval input. Bare interval operations are described; the accuracy of a decorated operation is defined to be that of its interval part.

12.10.1. *Measures of accuracy.* Three **accuracy modes** are defined that indicate the quality of interval enclosure achieved by an operation: *tightest*, *accurate* and *valid* in order from strongest to weakest. Each mode is in the first instance a property of an individual evaluation of an operation $\boldsymbol{f}$ of type $\mathbb{T}$ over an input box $\boldsymbol{x}$. The term **tightness** means the strongest mode that holds uniformly for some set of evaluations, e.g., for some one-argument function, an implementation might document the tightness of $\boldsymbol{f}(\boldsymbol{x})$ as being *tightest* for all $\boldsymbol{x}$ contained in $[-10^{15}, 10^{15}]$ and at least *accurate* for all other $\boldsymbol{x}$.

The *tightest* and *valid* modes apply to all interval types and all operations. 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 §10.5.3, 10.5.5 (because it 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 $\boldsymbol{f}_{\text{exact}}$ denote the corresponding Level 1 operation. The weakest mode *valid* is just the property of enclosure:

$$\boldsymbol{f}(\boldsymbol{x}) \supseteq \boldsymbol{f}_{\text{exact}}(\boldsymbol{x}). \tag{26}$$

For a forward arithmetic operation, it is equivalent to (25).

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

$$\boldsymbol{f}_{\text{tightest}}(\boldsymbol{x}) = \operatorname{hull}_{\mathbb{T}}(\boldsymbol{f}_{\text{exact}}(\boldsymbol{x})). \tag{27}$$

The intermediate mode *accurate* applies to an inf-sup type $\mathbb{T}$ derived from a number format $\mathbb{F}$. It asserts that $\boldsymbol{f}(\boldsymbol{x})$ is *valid*, (26), and is at most slightly wider than the result of applying the *tightest* version to a slightly wider input box:

$$\boldsymbol{f}(\boldsymbol{x}) \subseteq \texttt{nextOut}(\boldsymbol{f}_{\text{tightest}}(\texttt{nextOut}(\operatorname{hull}_{\mathbb{T}}(\boldsymbol{x})))). \tag{28}$$

Here the `nextOut` function is defined in terms of the functions

$$\texttt{nextUp}, \texttt{nextDown} : \operatorname{Val}(\mathbb{F}) \to \operatorname{Val}(\mathbb{F})$$

as follows. For $x \in \operatorname{Val}(\mathbb{F})$, define `nextUp`$(x)$ to be $+\infty$ if $x = +\infty$, and the least member of $\operatorname{Val}(\mathbb{F})$ greater than x otherwise; since $+\infty \in \operatorname{Val}(\mathbb{F})$, this is well-defined. Define `nextDown`(x) to be $-$`nextUp`$(-x)$. Then, if $\boldsymbol{x}$ is a $\mathbb{T}$-interval, define

$$\texttt{nextOut}(\boldsymbol{x}) = \begin{cases} [\,\texttt{nextDown}(\underline{x}), \texttt{nextUp}(\overline{x})\,] & \text{if } \boldsymbol{x} = [\underline{x}, \overline{x}] \neq \emptyset, \\ \emptyset & \text{if } \boldsymbol{x} = \emptyset. \end{cases} \tag{29}$$

When $\boldsymbol{x}$ is a $\mathbb{T}$-box, `nextOut` acts componentwise.
[*Notes.*

- *For a 754 format,* `nextUp` *and* `nextDown` *are equivalent to the corresponding functions in 754-2008 §5.3.1.*
- *In (28), the inner* `nextOut()` *aims to handle the problem of a function such as* $\sin x$ *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 might be hard to achieve for some special functions at some arguments.*
- *The input box* $\boldsymbol{x}$ *might have components of a different type from the result type* $\mathbb{T}$*, in the case of 754-conforming mixed-type operations §12.6.2. The* $\mathsf{hull}_{\mathbb{T}}$ *in (28) forces these to have type* $\mathbb{T}$*, so each component of* $\boldsymbol{x}$ *is widened by at least the local spacing of* $\mathbb{F}$*-numbers, at each finite bound.*

  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* $[\texttt{nextDown}(2.401), \texttt{nextUp}(3.699)] = [2.4, 3.7]$*, giving an insignificant widening. But*

$$\texttt{nextOut}(\mathsf{hull}_{\mathbb{T}}([2.401, 3.699])) = \texttt{nextOut}([2.4, 3.7]) = [2.3, 3.8]$$

  gives a widening comparable with the precision of $\mathbb{T}$*.*

]

12.10.2. *Accuracy requirements.* Following the categories of functions in Table 9.1, the accuracy of the *basic operations*, the *integer functions* and the *absmax functions* shall be *tightest*.

For all other operations in Table 9.1, for the reverse mode operations of Table 10.1, and for the recommended operations of Table 10.5 and Table 10.6, the accuracy shall be *valid*, and, for inf-sup types, should be *accurate*. For any operation in these four tables, if any input is Empty the result shall be Empty.

12.10.3. *Documentation requirements.* An implementation shall document the tightness of each of its interval operations for each supported bare interval type. This shall be done by dividing the set of possible inputs into disjoint subsets ("ranges") and stating a tightness achieved 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. Sample tightness information for the* sin *function might be*

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

]

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

12.11. Interval and number literals.

12.11.1. *Overview.*
This subclause defines an **interval literal**: a (text) string that denotes a bare or decorated interval. This entails defining a **number literal**: a string within an interval literal that denotes an extended-real number; if it denotes a finite integer it is an **integer literal**.

The bare or decorated interval denoted by an interval literal is its *value* (as an interval literal). Any other string has no value in that sense. For convenience a string is called *valid* or *invalid* (as an interval literal) according as it does or does not have such a value. The usage of value, valid and invalid for number and integer literals is similar.

Interval literals are used as input to `textToInterval` §12.12.7 and in the Input/Output clause §13. In this standard, number (and integer) literals are only used within interval literals. The definitions of literals are not intended to constrain the syntax and semantics that a language might use to denote numbers and intervals in other contexts.

The definition of an interval literal $\boldsymbol{s}$ is placed in Level 2 because its use is not mandatory at Level 1, see §10.5.1. However, the value of $\boldsymbol{s}$ is a bare or decorated Level 1 interval $\boldsymbol{x}$. Within $\boldsymbol{s}$, conversion of number literals to their values shall be done as if in infinite precision. Conversion of

$\boldsymbol{x}$ to a Level 2 datum $\boldsymbol{y}$ is a separate operation. In all cases $\boldsymbol{y}$ shall contain $\boldsymbol{x}$; typically $\boldsymbol{y}$ is the $\mathbb{T}$-hull of $\boldsymbol{x}$ for some interval type $\mathbb{T}$.
[*Example. The interval denoted by the literal* `[1.2345]` *is the Level 1 single-point interval* $\boldsymbol{x} = [1.2345, 1.2345]_{\texttt{com}}$. *However, the result of* $\mathbb{T}$`-textToInterval("[1.2345]")`*, where* $\mathbb{T}$ *is the 754 infsup* `binary64` *type, is the interval, approximately* $[1.2344999999999999, 1.2345000000000002]_{\texttt{com}}$*, whose bounds are the nearest* `binary64` *numbers either side of* 1.2345.]

The case of alphabetic characters in interval and number literals, including decorations, is ignored. It is assumed here that they have been converted to lowercase. (E.g., `inf` is equivalent to `Inf` and `[1,2]_dac` is equivalent to `[1,2]_DAC`.)

12.11.2. *Number literals.*

An integer literal comprises an optional sign and (i.e., followed by) a nonempty sequence of decimal digits.

The following forms of number literal shall be provided.

(a) A decimal number. This comprises an optional sign, a nonempty sequence of decimal digits optionally containing a point, and an optional exponent field comprising `e` and an integer literal. The value of a decimal number is the value of the sequence of decimal digits with optional point multiplied by ten raised to the power of the value of the integer literal, negated if there is a leading `-` sign.
(b) A number in the hexadecimal-floating-constant form of the C99 standard (ISO/IEC9899, N1256, §6.4.4.2), equivalently hexadecimal-significand form of IEEE 754-2008, §5.12.3. This comprises an optional sign, the string `0x`, a nonempty sequence of hexadecimal digits optionally containing a point, and an exponent field comprising `p` and an integer literal. The value of a hexadecimal number is the value of the sequence of hexadecimal digits with optional point multiplied by two raised to the power of the value of the integer literal, negated if there is a leading `-` sign.
(c) Either of the strings `inf` or `infinity` optionally preceded by `+`, with value $+\infty$; or preceded by `-`, with value $-\infty$.
(d) A rational literal p `/` q, that is p and q separated by the `/` character, where p, q are decimal integer literals, with q positive. Its value is the exact rational number p/q.

An implementation may support a more general form of integer and/or number literal, e.g., in the syntax of the host language of the implementation. It may restrict the support of literals, by relaxing conversion accuracy of hard cases: rational literals, long strings, etc., converting such literals to Entire, for example. It shall document such restrictions.

By default the syntax shall be that of the default locale (C locale); locale-specific variants may be provided.

12.11.3. *Unit in last place.* The "uncertain form" of interval literal, below, uses the notion of the *unit in the last place* of a number literal $\boldsymbol{s}$ of some radix b, possibly containing a point but without an exponent field. Ignoring the sign and any radix-specifying code (such as `0x` for hexadecimal), $\boldsymbol{s}$ is a nonempty sequence of radix-b digits optionally containing a point. Its *last place* is the integer $p = -d$ where $d = 0$ if $\boldsymbol{s}$ contains no point, otherwise d is the number of digits after the point. Then $\texttt{ulp}(\boldsymbol{s})$ is defined to equal b^p. When context makes clear, "x ulps of $\boldsymbol{s}$" or just "x ulps", is used to mean $x \times \texttt{ulp}(\boldsymbol{s})$. [*Example. For the decimal strings* `123` *and* `123.`*, as well as* `0` *and* `0.`*, the last place is* 0 *and one ulp is* 1. *For* `.123` *and* `0.123`*, as well as* `.000` *and* `0.000`*, the last place is* -3 *and one ulp is* 0.001.]

12.11.4. *Bare intervals.*

The following forms of bare interval literal shall be supported. 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 denotes zero or more space characters.

(a) Special values: The strings `[ ]` and `[ empty ]`, whose bare value is Empty; the string `[ entire ]`, whose bare value is Entire.
(b) Inf-sup form: A string `[` l `,` u `]` where l and u are optional number literals with $l \le u$, $l < +\infty$ and $u > -\infty$, see §10.2 and the note on difficulties of implementation §12.12.7. Its bare value is the mathematical interval $[l, u]$. Any of l and u may be omitted, with implied values $l = -\infty$ and $u = +\infty$, respectively. A string `[` m `]` with number literal m is equivalent to `[` m `,` m `]`.

Form	Literal	Exact decorated value
Special	`[ ]`	$\text{Empty}_{\texttt{trv}}$
	`[entire]`	$[-\infty,+\infty]_{\texttt{dac}}$
Inf-sup	`[1.e-3, 1.1e-3]`	$[0.001, 0.0011]_{\texttt{com}}$
	`[-Inf, 2/3]`	$[-\infty, 2/3]_{\texttt{dac}}$
	`[0x1.3p-1,]`	$[19/32, +\infty]_{\texttt{dac}}$
	`[,]`	$[-\infty,+\infty]_{\texttt{dac}}$
Uncertain	`3.56?1`	$[3.55, 3.57]_{\texttt{com}}$
	`3.56?1e2`	$[355, 357]_{\texttt{com}}$
	`3.560?2`	$[3.558, 3.562]_{\texttt{com}}$
	`3.56?`	$[3.555, 3.565]_{\texttt{com}}$
	`3.560?2u`	$[3.560, 3.562]_{\texttt{com}}$
	`-10?`	$[-10.5, -9.5]_{\texttt{com}}$
	`-10?u`	$[-10.0, -9.5]_{\texttt{com}}$
	`-10?12`	$[-22.0, 2.0]_{\texttt{com}}$
	`-10??u`	$[-10.0, +\infty]_{\texttt{dac}}$
	`-10??`	$[-\infty,+\infty]_{\texttt{dac}}$
NaI	`[nai]`	$\text{Empty}_{\texttt{ill}}$
Decorated	`3.56?1_def`	$[3.55, 3.57]_{\texttt{def}}$

TABLE 12.1. Portable interval literal examples.

(c) Uncertain form: a string m `?` r u E where: m is a decimal number literal of form (a) above, without exponent; r is empty or is a non-negative decimal integer literal *ulp-count* or is the `?` character; u is empty or is a *direction character*, either `u` (up) or `d` (down); and E is empty or is an *exponent field* comprising the character `e` followed by a decimal integer literal *exponent* e. No whitespace is permitted within the string.

With `ulp` meaning `ulp`(m), the literal m`?` by itself denotes m with a symmetrical uncertainty of half an 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` (down) or `u` (up) 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 $[m, m + r \times \texttt{ulp}]$. Uncertain form with radius empty or *ulp-count* is adequate for narrow (and hence bounded) intervals, but is severely restricted otherwise. Uncertain form with radius `?` is for unbounded intervals, e.g., m`??d` denotes $[-\infty, m]$, m`??u` denotes $[m, +\infty]$ and m`??` denotes Entire with m being like a comment. The exponent field if present multiplies the whole interval by 10^e, e.g., m `?`r`u` `e`e denotes $10^e \times [m, m + r \times \texttt{ulp}]$.

12.11.5. *Decorated intervals.*

A decorated interval literal may denote either a bare or a decorated interval value depending on context. The following forms of decorated interval literal shall be supported.

(a) The string `[ nai ]`, with the bare value Empty and the decorated value $\text{Empty}_{\texttt{ill}}$.

(b) A bare interval literal $\boldsymbol{sx}$.

If $\boldsymbol{sx}$ has the bare value $\boldsymbol{x}$, then $\boldsymbol{sx}$ has the decorated value `newDec`$(\boldsymbol{x})$ §11.5. Otherwise $\boldsymbol{sx}$ has no decorated value.

(c) A bare interval literal $\boldsymbol{sx}$, an underscore "`_`", and a 3-character decoration string $\boldsymbol{sd}$; where $\boldsymbol{sd}$ is one of `trv`, `def`, `dac` or `com`, denoting the corresponding decoration dx.

If $\boldsymbol{sx}$ has the bare value $\boldsymbol{x}$, and if $\boldsymbol{x}_{dx}$ is a permitted combination according to §11.4, then $\boldsymbol{s}$ has the bare value $\boldsymbol{x}$ and the decorated value $\boldsymbol{x}_{dx}$. Otherwise $\boldsymbol{s}$ has no value as a decorated interval literal.

[*Examples. Table 12.1 illustrates valid portable interval literals. These strings are not valid portable interval literals:* `empty`, `[5?1]`, `[1_000_000]`, `[ganz]`, `[entire!comment]`, `[inf]`, `5???u`, `[nai]_ill`, `[]_ill`, `[]_def`, `[0,inf]_com`.]

12.11.6. *Grammar for portable literals.*

Portable literals permit exchange between different implementations.

decDigit	[0123456789]
nonzeroDecDigit	[123456789]
hexDigit	[0123456789abcdef]
spaceChar	[\t]
natural	{decDigit} +
sign	[+-]
integerLiteral	{sign} ? {natural}
decSignificand	{decDigit} * "." {decDigit} + | {decDigit} + "." | {decDigit} +
hexSignificand	{hexDigit} * "." {hexDigit} + | {hexDigit} + "." | {hexDigit} +
decNumLit	{sign} ? {decSignificand} ("e" {integerLiteral}) ?
hexNumLit	{sign} ? "0x" {hexSignificand} "p" {integerLiteral}
infNumLit	{sign} ? ("inf" | "infinity")
positiveNatural	("0") * {nonzeroDecDigit} {decDigit} *
ratNumLit	{integerLiteral} "/" {positiveNatural}
numberLiteral	{decNumLit} | {hexNumLit} | {infNumLit} | {ratNumLit}
sp	{spaceChar} *
dir	"d" | "u"
pointIntvl	"[" {sp} {numberLiteral} {sp} "]"
infSupIntvl	"[" {sp} {numberLiteral} ? {sp} "," {sp} {numberLiteral} ? {sp} "]"
radius	{natural} | "?"
uncertIntvl	{sign} ? {decSignificand} "?" {radius} ? {dir} ? ("e" {integerLiteral}) ?
emptyIntvl	"[" {sp} "]" | "[" {sp} "empty" {sp} "]"
entireIntvl	"[" {sp} "entire" {sp} "]"
specialIntvl	{emptyIntvl} | {entireIntvl}
bareIntvlLiteral	{pointIntvl} | {infSupIntvl} | {uncertIntvl} | {specialIntvl}
NaI	"[" {sp} "nai" {sp} "]"
decorationLit	"trv" | "def" | "dac" | "com"
intervalLiteral	{NaI} | {bareIntvlLiteral} | {bareIntvlLiteral} "_" {decorationLit}

TABLE 12.2. Grammar for literals: integer literal is integerLiteral, number literal is numberLiteral, bare interval literal is bareIntvlLiteral and decorated interval literal is intervalLiteral.

The syntax of portable integer and number literals and of portable bare and decorated interval literals is defined by integerLiteral, numberLiteral, bareIntvlLiteral and intervalLiteral, respectively, in the grammar in Table 12.2, which uses the notation of 754 §5.12.3. Lowercase is assumed, i.e., a valid string is one that after conversion to lowercase is accepted by this grammar. `\t` denotes the TAB character.

The constructor `textToInterval` §12.12.7, 13.2 of any implementation shall accept any portable interval. An implementation may restrict support of some input strings (too long strings or strings with a rational number literal). Nevertheless, the constructor shall always return a Level 2 interval (possibly Entire in this case) that contains the Level 1 interval.

An implementation may support interval literals of more general syntax (for example, with underscores in significand). In this case there shall be a value of conversion specifier *cs* that restricts output strings of `intervalToText` §13.3 to the portable syntax.

12.12. Required operations on bare and decorated intervals.

An implementation shall provide a $\mathbb{T}$-version, see §12.9, of each operation listed in §12.12.1 to §12.12.10, for each supported type $\mathbb{T}$. That is, those of the $\mathbb{T}$-version's inputs and outputs that are intervals are of type $\mathbb{T}$ (or the corresponding decorated type), except for the conversion operation of §12.12.10 whose output is of type $\mathbb{T}$ and whose input is of any type.

The implementation shall provide the type-independent decoration operations of §12.12.11. It shall provide the reduction operations of §12.12.12 for the parent formats of supported 754-conforming types.

Operations in this subclause are described as functions with zero or more input arguments and one return value. It is language- or implementation-defined whether they are implemented in this way: for instance, two-output division, described in §12.12.3 as a function returning an ordered pair of intervals, might be implemented as a procedure $\mathtt{divToPair}(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}_1, \boldsymbol{z}_2)$ with input arguments $\boldsymbol{x}$ and $\boldsymbol{y}$ and output arguments $\boldsymbol{z}_1$ and $\boldsymbol{z}_2$.

An implementation, or a part thereof, that is 754-conforming shall provide mixed-type operations, as specified in §12.6.2, for the following operations, which correspond to those that 754 requires to be provided as *formatOf* operations.

`add, sub, mul, div, recip, sqrt, sqr, 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.

12.12.1. *Interval constants.* For each supported bare interval type $\mathbb{T}$ there shall be a $\mathbb{T}$-version of each constant function `empty`() and `entire`() of §10.5.2, returning a $\mathbb{T}$-interval with value Empty and Entire, respectively. There shall also be a decorated version of each, returning `newDec`(Empty) = $\text{Empty}_{\mathtt{trv}}$ and `newDec`(Entire) = $\text{Entire}_{\mathtt{dac}}$, respectively, of the derived decorated type.

12.12.2. *Forward-mode elementary functions.* Let $\mathbb{T}$ be a supported bare interval type and $\mathbb{DT}$ the derived decorated type. An implementation shall provide a $\mathbb{T}$-version of each forward arithmetic operation in §10.5.3. Its inputs and output are $\mathbb{T}$-intervals, and it shall be a Level 2 interval extension of the corresponding point function. Recommended accuracies are given in §12.10.
[*Note. For operations, some of whose arguments are of integer type, such as integer power* $\mathtt{pown}(x, p)$*, only the real arguments are replaced by intervals.*]

Each such operation shall have a decorated version with corresponding arguments of type $\mathbb{DT}$. It shall be a decorated interval extension as defined in §11.6—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 (17) of §11.6, 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 (17).

12.12.3. *Two-output division.*

There shall be a $\mathbb{T}$-version of the two-output division $\mathtt{divToPair}(\boldsymbol{x}, \boldsymbol{y})$ of §10.5.4, for each supported bare interval type $\mathbb{T}$, namely

$$(\boldsymbol{u}, \boldsymbol{v}) = \mathtt{divToPair}(\boldsymbol{x}, \boldsymbol{y}),$$

where $\boldsymbol{x}$ and $\boldsymbol{y}$ are $\mathbb{T}$-intervals. Each of the outputs $\boldsymbol{u}$ and $\boldsymbol{v}$ is a $\mathbb{T}$-interval that encloses the corresponding Level 1 value.

There shall be a decorated version where each of $\boldsymbol{x}$, $\boldsymbol{y}$, $\boldsymbol{u}$ and $\boldsymbol{v}$ is of the corresponding decorated type. If either input is NaI then both outputs are NaI. Otherwise, if $\boldsymbol{x}$ and $\boldsymbol{y}$ are nonempty and $0 \notin \boldsymbol{y}$, then $\boldsymbol{u}$ is the same as the result of normal division $\boldsymbol{x}/\boldsymbol{y}$ and shall be decorated the same way; while $\boldsymbol{v}$ is empty and shall be decorated `trv`. In all other cases each output, empty or not, shall be decorated `trv`.

12.12.4. *Reverse-mode elementary functions.* An implementation shall provide a $\mathbb{T}$-version of each reverse arithmetic operation in §10.5.5, for each supported bare interval type $\mathbb{T}$. Its inputs and output are $\mathbb{T}$-intervals.

These operations shall have "trivial" decorated versions, as described in §11.7.

12.12.5. *Cancellative addition and subtraction.* An implementation shall provide a $\mathbb{T}$-version of each of the operations `cancelMinus` and `cancelPlus` in §10.5.6 for each supported bare interval type $\mathbb{T}$. Its inputs and output are $\mathbb{T}$-intervals.

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

– either of $\boldsymbol{x}$ and $\boldsymbol{y}$ is unbounded;
– $\boldsymbol{x} \neq \emptyset$ and $\boldsymbol{y} = \emptyset$;

– $\boldsymbol{x}$ and $\boldsymbol{y}$ are nonempty bounded intervals with $\texttt{width}(\boldsymbol{x}) < \texttt{width}(\boldsymbol{y})$.

`cancelPlus`$(\boldsymbol{x}, \boldsymbol{y})$ shall be equivalent to `cancelMinus`$(\boldsymbol{x}, -\boldsymbol{y})$.

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.

These operations shall have "trivial" decorated versions, as described in §11.7.

[*Notes. Two cases need care: see §B4 for more detail. Examples are given where $\mathbb{T}$ is the inf-sup type of a format $\mathbb{F}$.*

- *Determining whether the Level 1 value is defined needs care when $\boldsymbol{x}$ and $\boldsymbol{y}$ have equal widths in finite precision. An example is* `cancelMinus`$([-1, a], [-b, 1])$ *where a and b are positive $\mathbb{F}$-numbers less than $\mathbb{F}$'s roundoff unit.*
- *For any $\boldsymbol{x}, \boldsymbol{y}$, the Level 1 result is always bounded when it is defined, but might overflow at Level 2. An example is* `cancelMinus`$([M, M], [-M, -M])$ *where M is the largest finite $\mathbb{F}$-number. The exact value is $[2M, 2M]$, whose $\mathbb{T}$-hull is $[M, +\infty]$. An implementation should not return* Entire *in case of overflow. For any inf-sup type it can avoid doing so, since the Level 1 result has width not exceeding that of $\boldsymbol{x}$, and therefore at Level 2 cannot overflow to both $-\infty$ and $+\infty$. Not returning* Entire *in case of overflow makes the result* Entire *diagnostic: it occurs if, and only if, there is no value at Level 1.*

]

12.12.6. *Set operations.* An implementation shall provide a $\mathbb{T}$-version of each of the operations `intersection` and `convexHull` in §10.5.7 for each supported bare interval type $\mathbb{T}$. Its inputs and output are $\mathbb{T}$-intervals.

These operations should return the $\mathbb{T}$-hull of the exact result. If either input to `intersection` is Empty, or both inputs to `convexHull` are Empty, the result shall be Empty.

[*Note. If $\mathbb{T}$ is an inf-sup type, the operation can always return the exact result.*]

These operations shall have "trivial" decorated versions, as described in §11.7.

12.12.7. *Constructors.* For each supported bare or decorated interval type $\mathbb{T}$, there shall be a $\mathbb{T}$-version of each constructor in §10.5.8. It returns a $\mathbb{T}$-datum.

Difficulties in implementation. Both `textToInterval` when its input is a literal of inf-sup form, and `numsToInterval`, involve testing if a boolean value $b = (l \leq u)$ is 0 (false) or 1 (true), to determine whether the interval is empty or nonempty. Here l and u are extended-real numbers. In the former case they are values of number literals; in the latter, they are values of datums of supported number formats.

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—a containment failure. 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.

[*Note. Evaluating b correctly can be hard, if finite 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 values of different radices. It could be especially challenging in an extendable-precision context.*

Language rules might cause such errors even when l and u are of the same number 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`*, on entry to a* `numsToInterval` *call. This would be non-conforming—the comparison $l \leq u$ requires the exact values to be used, which requires use of a version of* `numsToInterval` *with* `long double` *arguments.*]

Bare interval constructors. A bare interval constructor call either **succeeds** or **fails**. This notion is used to determine the value returned by the corresponding decorated interval constructor.

For the constructor `numsToInterval`(l, u), the inputs l and u are datums of supported number formats, where the format of l may differ from that of u. For a given bare interval type $\mathbb{T}$:

- There shall be a version of `numsToInterval` where l and u are both of the same format, compatible with $\mathbb{T}$, see §12.4.
- If $\mathbb{T}$ is 754-conforming, there shall be a *formatOf* version of `numsToInterval` that accepts l and u having any combination of supported 754 formats of the same radix as $\mathbb{T}$.

Apart from these two requirements, what (l, u) format combinations are provided is language- or implementation-defined.

The constructor call succeeds if (a) the implementation determines that the call has a Level 1 value $\boldsymbol{x}$, see §10.5.8, or (b) it cannot determine whether a Level 1 value exists, see the discussion at the start of this subclause. The conditions under which case (b) might occur shall be documented.

Case (a) is that neither l nor u is NaN, and the exact extended-real values they denote—also called l and u for brevity—are known to satisfy $l \leq u$, $l < +\infty$, $u > -\infty$. In this case the result shall be a $\mathbb{T}$-interval containing $\boldsymbol{x}$. In case (b) the result shall be a $\mathbb{T}$-interval containing l and u.

If $\mathbb{T}$ is a 754-conforming type and l and u are of 754 formats with the same radix as $\mathbb{T}$, case (b) cannot arise, and the result shall be the $\mathbb{T}$-hull of $\boldsymbol{x}$; in particular if $\boldsymbol{x}$ is an exact $\mathbb{T}$-interval, the result is $\boldsymbol{x}$. For other cases, the tightness of the result is implementation-defined.

Otherwise the call fails, and the result is Empty.

For the constructor `textToInterval`$(\boldsymbol{s})$, the input $\boldsymbol{s}$ is a string. The constructor call succeeds if: (a) the implementation determines that $\boldsymbol{s}$ is a valid interval literal with bare value $\boldsymbol{x}$, see §12.11; or (b) $\boldsymbol{s}$ is of inf-sup form `[`l`,`u`]` or `[`l`,`u`]_`dx with finite l and u, but the implementation cannot determine whether a Level 1 value exists, i.e. whether $l \leq u$. The conditions under which case (b) might occur shall be documented.

In case (a) the result shall be a $\mathbb{T}$-interval containing $\boldsymbol{x}$. If $\mathbb{T}$ is a 754-conforming type, this shall be the $\mathbb{T}$-hull of $\boldsymbol{x}$; in particular if $\boldsymbol{x}$ is an exact $\mathbb{T}$-interval, the result is $\boldsymbol{x}$. For other types $\mathbb{T}$, the tightness of the result is implementation-defined. In case (b) the result shall be an interval containing l and u.

Otherwise the call fails, and the result is Empty.

Decorated interval constructors. Each bare interval constructor shall have a corresponding decorated constructor, taking the same input(s) as the bare constructor. The decorated constructor succeeds if and only if the bare interval constructor succeeds. The decorated constructor fails returning NaI if and only if the bare interval constructor fails.

If the bare interval constructor `numsToInterval`(l, u) succeeds, returning $\boldsymbol{y}$, the decorated form returns `newDec`$(\boldsymbol{y})$, see §11.5.

If the bare interval constructor `textToInterval`$(\boldsymbol{s})$ succeeds, returning $\boldsymbol{y}$, the decorated form returns $\boldsymbol{y}_{dy}$, where dy is determined as follows:

– when $\boldsymbol{s}$ is a valid interval literal with decorated value $\boldsymbol{x}_{dx}$, then dy shall equal dx, except in the case that $dx =$ `com` and overflow occurred, that is, $\boldsymbol{x}$ is bounded and $\boldsymbol{y}$ is unbounded. Then dy shall equal `dac`.

Exception behavior. `UndefinedOperation` is signaled by both the bare and the decorated constructor when the input is such that the bare constructor fails. [*Note. When signaled by the decorated constructor it will normally be ignored since returning* NaI *gives sufficient information.*] `PossiblyUndefinedOperation` is signaled by both the bare and the decorated constructor when the input is such that the implementation cannot determine whether a Level 1 value exists (the two cases (b) above).

12.12.8. *Numeric functions of intervals.* An implementation shall provide a $\mathbb{T}$-version of each numeric function in Table 10.2 of §10.5.9 for each supported bare interval type $\mathbb{T}$, giving a result in a supported number format $\mathbb{F}$ that should be compatible with $\mathbb{T}$, see §12.4. An implementation may provide several versions, returning results in different formats. If $\mathbb{T}$ is a 754-conforming type, versions shall be provided giving a result in any supported 754 format of the same radix as $\mathbb{T}$.

The mapping of a Level 1 value to an $\mathbb{F}$-number is defined in terms of the following rounding methods, which are functions from $\overline{\mathbb{R}}$ to $\mathbb{F}$. [*Note. These functions help define operations of the standard but are not themselves operations of the standard.*]

– *Round toward positive*: x maps to the smallest $\mathbb{F}$-number not less than x. If $\mathbb{F}$ has signed zeros, 0 maps to +0.
– *Round toward negative*: x maps to the largest $\mathbb{F}$-number not greater than x. If $\mathbb{F}$ has signed zeros, 0 maps to −0.
– *Round to nearest*: x maps to the $\mathbb{F}$-number (possibly $\pm\infty$) closest to x, 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. If $\mathbb{F}$ has signed zeros, 0 maps to +0.

The implementation shall document how it handles cases not covered by the above rules, e.g., the distance to an infinity and the method of tie-breaking. 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.

In formats that have a signed zero, a Level 1 value of 0 shall be returned as -0 by `inf`, and $+0$ by all other functions in this subclause.

- `inf`$(\boldsymbol{x})$ returns the Level 1 value rounded toward negative.
- `sup`$(\boldsymbol{x})$ returns the Level 1 value rounded toward positive.
- `mid`$(\boldsymbol{x})$: the result m is defined by the following cases, where $\underline{x}, \overline{x}$ are the exact (Level 1) lower and upper bounds of $\boldsymbol{x}$.

$\boldsymbol{x} =$ Empty	NaN.
$\boldsymbol{x} =$ Entire	0.
$\underline{x} = -\infty$, $\overline{x}$ 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.
$\underline{x}, \overline{x}$ both finite	m should be, and if $\mathbb{T}$ is a 754-conforming type shall be, the Level 1 value rounded to nearest.

The implementation shall document how it handles the last case.
[*Note. If $\mathbb{F}$ is compatible with $\mathbb{T}$, a nonempty $\boldsymbol{x}$ always contains m. If $\mathbb{F}$ is not compatible with $\mathbb{T}$ this might be impossible. E.g., let $\mathbb{T}$ be inf-sup* `binary64`*, and let $\boldsymbol{x} = [1+u, 1+2u]$ where u is the* `binary64` *roundoff unit. If $\mathbb{F}$ is* `binary64` *(compatible) then m is $1+u$ or $1+2u$ depending on the tie-breaking method. If $\mathbb{F}$ is* `binary32` *(incompatible) then no $\mathbb{F}$-numbers lie in $\boldsymbol{x}$, and $m = 1$.*]

- `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`$(\boldsymbol{x})$ *might 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 NaN if $\boldsymbol{x}$ is empty. Otherwise it returns the Level 1 value rounded toward +oo.
[*Note. For nonempty bounded $\boldsymbol{x}$ the ratio* `wid`$(\boldsymbol{x})/$`rad`$(\boldsymbol{x})$*, which is always 2 at Level 1, ranges from 1 to $+\infty$ at Level 2. When $\mathbb{F}$ is a 754 format and $\mathbb{T}$ is the derived inf-sup type, it takes the value 1 if the bounds of $\boldsymbol{x}$ are adjacent subnormal numbers, and the value $+\infty$ if $\boldsymbol{x} = [-$*`realmax`*, *`realmax`*$]$.*]
- `mag`$(\boldsymbol{x})$ returns NaN if $\boldsymbol{x}$ is empty. Otherwise it returns the Level 1 value rounded toward positive.
- `mig`$(\boldsymbol{x})$ returns NaN if $\boldsymbol{x}$ is empty. Otherwise it returns the Level 1 value rounded toward negative, except that 0 maps to $+0$ if $\mathbb{F}$ has signed zeros.

Each bare interval operation in this subclause shall have a decorated version, where each input of bare interval type is replaced by an input having the corresponding decorated interval type, and the result format is that of the bare operation. Following §11.7, if any input is NaI, the result is NaN. Otherwise the result is obtained by discarding the decoration and applying the corresponding bare interval operation.

12.12.9. *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 §10.5.10, for each supported bare interval type $\mathbb{T}$.

There shall be a function `isNaI`$(\boldsymbol{x})$ for input $\boldsymbol{x}$ of any decorated type, that returns `true` if $\boldsymbol{x}$ is NaI, else `false`.

There shall be a $\mathbb{T}$-version of each of the comparison relations in Table 10.3 of §10.5.10, 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 types of the same radix.

These comparisons shall return, in all cases, the correct value of the comparison applied to the intervals denoted by the inputs as if in infinite precision. In particular `equal`$(\boldsymbol{x}, \boldsymbol{y})$, for those bare interval inputs $\boldsymbol{x}$ and $\boldsymbol{y}$ for which it is defined, shall return `true` if and only if $\boldsymbol{x}$ and $\boldsymbol{y}$ (ignoring their type) are the same mathematical interval, see §12.3.

Each bare interval operation in this subclause shall have a decorated version, where each input of bare interval type is replaced by an input having the corresponding decorated interval type.

Following §11.7, if any input is NaI, the result is `false` (in particular `equal`(NaI, NaI) is `false`). Otherwise the result is obtained by discarding the decoration and applying the corresponding bare interval operation.

12.12.10. *Interval type conversion.* An implementation shall provide for each supported bare interval type $\mathbb{T}$ the operation $\mathbb{T}$-`convertType` to convert an interval of any supported bare interval type to type $\mathbb{T}$ and from any supported decorated interval type to the corresponding decorated type $\mathbb{DT}$. Conversion is done by applying the $\mathbb{T}$-hull operation, see §12.8.1. For a bare interval $\boldsymbol{x}$:

$$\mathbb{T}\text{-}\mathtt{convertType}(\boldsymbol{x}) = \mathrm{hull}_{\mathbb{T}}(\boldsymbol{x}).$$

Thus if $\mathbb{T}$ is an explicit type, see §12.8.1, the result is the unique tightest $\mathbb{T}$-interval containing $\boldsymbol{x}$.

Conversion of a decorated interval is done by converting the interval part, except that if the decoration is `com` and the conversion overflows (produces an unbounded interval) the decoration becomes `dac`. That is, $\mathbb{T}\text{-}\mathtt{convertType}(\boldsymbol{x}_{dx}) = \boldsymbol{y}_{dy}$ where

$$\begin{aligned} \boldsymbol{y} &= \mathbb{T}\text{-}\mathtt{convertType}(\boldsymbol{x}); \\ dy &= \begin{cases} \mathtt{dac} & \text{if } dx = \mathtt{com} \text{ and } \boldsymbol{y} \text{ is unbounded,} \\ dx & \text{otherwise.} \end{cases} \end{aligned}$$

12.12.11. *Operations on/with decorations.*

An implementation shall provide the operations of §11.5. These comprise the comparison operations $=, \neq, >, <, \geq, \leq$ for decorations; and, for each supported bare interval type and corresponding decorated type, the operations `newDec`, `intervalPart`, `decorationPart` and `setDec`.

A call `intervalPart`(NaI), whose value is undefined at Level 1, shall return Empty at Level 2, and shall signal the `IntvlPartOfNaI` exception to indicate that a valid interval has been created from the ill-formed interval.

12.12.12. *Reduction operations.* For each supported 754-conforming interval type, an implementation shall provide, for the parent format of that type, the four reduction operations `sum`, `dot`, `sumSquare` and `sumAbs` of IEEE 754-2008 §9.4, correctly rounded.

Correctly rounded means that the returned result is defined as follows.

– If the exact result is defined as an extended-real number, return this after rounding to the relevant format according to the current rounding direction. An exact zero shall be returned as +0 in all rounding directions, except for roundTowardNegative, where –0 shall be returned.
– For `dot` and `sum`, if a NaN is encountered, or if infinities of both signs were encountered in the sum, NaN shall be returned. ("NaN encountered" includes the case $\infty \times 0$ for `dot`.)
– For `sumAbs` and `sumSquare`, if an Infinity is encountered, $+\infty$ shall be returned. Otherwise, if a NaN is encountered, NaN shall be returned.

(These rules allow for short-circuit evaluation in certain cases.)

All other behavior, such as overflow, underflow, setting of IEEE 754 flags, signaling exceptions, and behavior on vectors whose length is given as non-integral, zero or negative, shall be as specified in IEEE 754-2008 §9.4. In particular, evaluation is as if in exact arithmetic up to the final rounding, with no possibility of intermediate overflow or underflow.

Also, since correct rounding applies, the Inexact flag shall be set unless an exact extended-numeric result is returned. (If a final overflow or underflow is indicated, the result is inexact.)

Intermediate overflow could result from adding an extremely large number N of large terms of the same sign. The implementation shall ensure this cannot occur. This is done by providing enough leading carry bits in an accumulator, or equivalent, so that the N required is unachievable with current hardware.

[*Note. For example, Complete Arithmetic for IEEE 754* `binary64`*, parameterized as recommended by Kulisch and Snyder, requires around* 2^{88} *terms before overflow can occur.*]

It is recommended that these operations be based on an implementation of Complete Arithmetic as specified in §B3.

12.13. Recommended operations.

12.13.1. *Forward-mode elementary functions.*

The functions listed in §10.6.1 are arithmetic operations. If any of them is provided, it shall have a version for each bare and decorated interval type, specified as is done in §12.12.2 for the required operations.

12.13.2. *Slope functions.*

The functions listed in §10.6.2 shall be handled in the same way as those in §12.13.1.

12.13.3. *Boolean functions of intervals.*

The functions listed in §10.6.3 shall be handled in the same way as those in §12.13.1.

If $\mathbb{T}$-version of the function `isMember`$(m, \boldsymbol{x})$ is provided, the number format $\mathbb{F}$ of the argument m should be compatible with $\mathbb{T}$. An implementation may provide several $\mathbb{T}$-versions, with different formats of m. If $\mathbb{T}$ is a 754-conforming type, versions should be provided with the argument m of any supported 754 format of the same radix as $\mathbb{T}$.

12.13.4. *Extended interval comparisons.*

How the operations in §10.6.4 are handled at Level 2 is implementation-defined.

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

13.1. Overview. This clause of the standard specifies conversion from a text string that holds an interval literal to an interval internal to a program (input), and the reverse (output). The methods by which strings are read from, or written to, a character stream are language- or implementation-defined, as are variations in some locales (such as specific character case matching).

Containment is preserved on input and output so that, when a program computes an enclosure of some quantity given an enclosure of the data, it can ensure this holds all the way from text data to text results.

In addition to the above I/O, which might incur rounding errors on output and/or input, each interval type $\mathbb{T}$ has an *exact text representation*, via operations that convert any internal $\mathbb{T}$-interval $\boldsymbol{x}$ to a string $\boldsymbol{s}$, and back again to recover $\boldsymbol{x}$ exactly.

13.2. Input. Input is provided for each supported bare or decorated interval type $\mathbb{T}$ by the $\mathbb{T}$-version of `textToInterval`$(\boldsymbol{s})$, where $\boldsymbol{s}$ is a string, as specified in §12.12.7. It accepts an arbitrary interval literal $\boldsymbol{s}$ and returns a $\mathbb{T}$-interval enclosing the Level 1 value of $\boldsymbol{s}$.

For 754-conforming types $\mathbb{T}$ the required tightness is specified in §12.12.7. For other types, the tightness is implementation-defined.
[*Note. This provides the basis for free-format input of interval literals from a text stream, as might be provided by overloading the >> operator in C++.*]

13.3. Output. An implementation shall provide an operation

$$\texttt{intervalToText}(\boldsymbol{X}, cs),$$

where cs is optional. $\boldsymbol{X}$ is a bare or decorated interval datum of any supported interval type $\mathbb{T}$, and cs is a string, the conversion specifier. The operation converts $\boldsymbol{X}$ to a valid interval literal string $\boldsymbol{s}$, see§12.11, which shall be related to $\boldsymbol{X}$ as follows, where $\boldsymbol{Y}$ is the Level 1 value of $\boldsymbol{s}$.

(i) Let $\mathbb{T}$ be a bare type. Then $\boldsymbol{Y}$ shall contain $\boldsymbol{X}$ and shall be empty if $\boldsymbol{X}$ is empty.
(ii) Let $\mathbb{T}$ be a decorated type. If $\boldsymbol{X}$ is NaI, then $\boldsymbol{Y}$ shall be NaI. Otherwise, write $\boldsymbol{X} = \boldsymbol{x}_{dx}$, $\boldsymbol{Y} = \boldsymbol{y}_{dy}$. Then
- $\boldsymbol{y}$ shall contain $\boldsymbol{x}$ and shall be empty if $\boldsymbol{x}$ is empty.
- dy shall equal dx, except in the case that $dx =$ `com` and overflow occurred, that is, $\boldsymbol{x}$ is bounded and $\boldsymbol{y}$ is unbounded. Then dy shall equal `dac`.

[*Note.* $\boldsymbol{Y}$ *being a Level 1 value is significant. E.g., for a bare type* $\mathbb{T}$*, it is not allowed to convert* $\boldsymbol{X} = \emptyset$ *to the string* `garbage`*, even though converting* `garbage` *back to a bare interval at Level 2 by* $\mathbb{T}$`-textToInterval` *gives* $\emptyset$*, because* `garbage` *has no Level 1 value as a bare interval literal.*]

The tightness of enclosure of $\boldsymbol{X}$ by $\boldsymbol{Y}$ is language- or implementation-defined.

If present, cs lets the user control the layout of the string $\boldsymbol{s}$ in a language- or implementation-defined way. The implementation shall document the recognized values of cs and their effect; other values are called *invalid*.

If cs is invalid, or makes an unsatisfiable request for a given input $\boldsymbol{X}$, the output shall still be an interval literal whose value encloses $\boldsymbol{X}$. A language- or implementation-defined extension to interval literal syntax may be used to make it obvious that this has occurred. [*Example. Suppose, for uncertain form, that* m *is undefined or* r *is "unreasonably large". Then a string such as* `[Entire!uncertain form conversion error]` *might be produced. The implementation of* `textToInterval` *would need to accept this string as meaning the same as* `[Entire]`*. Such a string is not a portable literal, see §12.11.6.*]

Among the user-controllable features should be the following, where l, u are the interval bounds for inf-sup form, and m, r are the base point and radius for uncertain form, as defined in §12.11.

(i) It should be possible to specify the preferred overall field width (the length of $\boldsymbol{s}$), and whether output is in inf-sup or uncertain form.
(ii) It should be possible to specify how Empty, Entire and NaI are output, e.g., whether lower or upper case, and whether Entire becomes `[Entire]` or `[-Inf, Inf]`.
(iii) For l, u and m, it should be possible to specify the field width, and the number of digits after the point or the number of significant digits. For r, which is a non-negative integer ulp-count, it should be possible to specify the field width. There should be a choice of radix, at least between decimal and hexadecimal.

(iv) For uncertain form, it should be possible to select the default symmetric form, or the one sided (`u` or `d`) forms. It should be possible to choose whether an exponent field is absent (and m is output to a given number of digits after the point) or present (and m is output to a given number of significant digits).

(v) It should be possible to output the bounds of an interval without punctuation, e.g., `1.234  2.345` instead of `[1.234, 2.345]`. For instance, this might be a convenient way to write intervals to a file for use by another application.

If *cs* is absent, output should be in a general-purpose layout (analogous, e.g., to the `%g` specifier of `fprintf` in C). There should be a value of *cs* that selects this layout explicitly.
[*Note. This provides the basis for free-format output of intervals to a text stream, as might be provided by overloading the << operator in C++.*]

If an implementation supports more general syntax of interval literals than the portable syntax defined in §12.11.6, there shall be a value of *cs* that restricts output strings to the portable syntax.

If $\mathbb{T}$ is a 754-conforming bare type, there shall be a value of *cs* that produces behavior identical with that of `intervalToExact`, below. That is, the output is an interval literal that, when read back by $\mathbb{T}$-`textToInterval`, recovers the original datum exactly.

13.4. Exact text representation. For any supported bare interval type $\mathbb{T}$, an implementation shall provide operations `intervalToExact` and `exactToInterval`. Their purpose is to provide a portable exact representation of every bare interval datum as a string.

These operations shall obey the **recovery requirement**:

> For any $\mathbb{T}$-datum $\boldsymbol{x}$, the value $\boldsymbol{s} = \mathbb{T}$-`intervalToExact`($\boldsymbol{x}$) is a string, such that $\boldsymbol{y} = \mathbb{T}$-`exactToInterval`($\boldsymbol{s}$) is defined and equals $\boldsymbol{x}$.

[*Note. From §12.3, this is datum identicality: $\boldsymbol{x}$ and $\boldsymbol{y}$ have the same Level 1 value and the same type. They might differ at Level 3, e.g., a zero bound might be stored as −0 in one and +0 in the other.*]

If $\mathbb{T}$ is a 754-conforming type, the string $\boldsymbol{s}$ shall be an interval literal which, for nonempty $\boldsymbol{x}$, is of inf-sup type, with the lower and upper bounds of $\boldsymbol{x}$ converted as described in §13.4.1. For such $\boldsymbol{s}$, the operation `exactToInterval` is functionally equivalent to `textToInterval`.

If $\mathbb{T}$ is not 754-conforming, there are no restrictions on the form of the string $\boldsymbol{s}$ apart from the above recovery requirement. However, the representation should display, in a human-readable way, the exact values of the parameters of the type's mathematical description (e.g., m, $\underline{r}$ and $\overline{r}$ for a type that represents an interval in triplex form $[m + \underline{r}, m + \overline{r}]$).

The algorithm by which `intervalToExact` converts $\boldsymbol{x}$ to $\boldsymbol{s}$ is regarded as part of the definition of the type and shall be documented by the implementation.
[*Example. Writing a* `binary64` *floating-point datum exactly in hexadecimal-significand form passes the "readability" test since it displays the parameters sign, exponent and significand. Dumping its 64 bits as 16 hex characters does not.*]

Since `exactToInterval` creates an interval from non-interval data, it is a constructor similar to `textToInterval`. When its input is invalid, it shall return Empty and signal one of the exceptions `UndefinedOperation` or `PossiblyUndefinedOperation` as specified in §12.12.7.

13.4.1. *Conversion of 754 numbers to strings.* A 754 format $\mathbb{F}$ is defined by the parameters: b = the radix, 2 or 10; p = the number of digits in the significand (precision); $emax$ = the maximum exponent; $emin = 1 - emax$ = the minimum exponent (see 754-2008 §3.3).

A finite $\mathbb{F}$-number x can be represented $(-1)^s \times b^e \times m$ where $s = 0$ or 1, e is an integer, $emin \le e \le emax$, and m has a p-digit radix b expansion $d_0.d_1d_2 \ldots d_{p-1}$, where d_i is an integer digit $0 \le d_i < b$ (so $0 \le m < b$). As used within interval literals, x denotes a real number, with no distinction between −0 and +0. To make the representation unique, constraints are imposed in three mutually exclusive cases:

– A *normal* number, with $|x| \ge b^{emin}$, shall have $d_0 \ge 1$ (so $1 \le m < b$).
– A *subnormal* number, with $0 < |x| < b^{emin}$, shall have $e = emin$, which implies $d_0 = 0$ (so $0 < m < 1$).
– *Zero*, $x = 0$, shall have sign bit $s = 0$ and exponent $e = 0$ (and necessarily $m = 0$).

[*Note. For $b = 2$ the standard form used by 754 is the same as this, except for replacing zero by two signed zeros, with exponent $e = emin$. For $b = 10$, there is also the difference that 754 normal* *numbers have several representations if they need fewer than p digits in their expansion. The standard form above chooses the representation with smallest quantum, which is the unique one having $d_0 \neq 0$.*]

The rules given below for converting x to a string *xstr* allow user-, language- or implementation-defined choice while ensuring the values of s, m and e are easily found from *xstr* in each of these cases, even without knowledge of the format parameters p, *emax*, *emin*.

xstr is the concatenation of: a sign part *sstr*; a significand part *mstr*; and an exponent part *estr*. If $b = 2$, the hex-indicator `"0x"` is prefixed to *mstr*.

sstr is `"-"` or an optional `"+"`, as appropriate.

If $b = 10$, *mstr* is the (decimal) expansion $d_0.d_1d_2\ldots d_{p-1}$, optionally abbreviated by removing some or all trailing zeros. If this leaves no digits after the point, the point may be removed. If $b = 2$, *mstr* is formed from the (binary) expansion $d_0.d_1d_2\ldots d_{p-1}$, abbreviated in the same way, and then converted to a hexadecimal string $D_0.D_1\ldots$ (so necessarily D_0 is 1 if x is normal, 0 if x is subnormal or zero).

estr consists of `"e"` if $b = 10$, `"p"` if $b = 2$, followed by the exponent e written as a signed decimal integer, with the sign optional if $e \geq 0$.

[*Examples. In any binary format, the number* 2 *(with* $s = 0$, $m = 1$, $e = 1$*) may be written as* `0x1p1` *or* `+0x1.0p+01`*, etc., but not as* `0x2p0`*; while* $\frac{1}{2}$ *may be written as* `0x1p-1` *or* `+0x1.0p-01`*, etc. The number* -4095 *(with* $s = 1$, $m = \frac{4095}{2048}$, $e = 11$*) may be written as* `-0x1.ffep+11`.

In `decimal32` *(see 754-2008 Table 3.6), which has* $p = 7$*, the smallest positive normal number may be written* `1e-95` *or* `+1.000000e-95`*, etc.; and the next number below it as* `0.999999e-95`*. The smallest positive number may be written* `0.000001e-95`.]

Above, alphabetic characters have been written in lowercase, but they may be in either case.

A shorter form for subnormal numbers may be used, normalized by requiring $d_1 \neq 0$; however, to find the canonical m and e from *xstr* one then needs to know *emin*. For instance, the smallest positive `decimal32` number $x =$ `0.000001e-95` has the shorter form `0.1e-101`, but to deduce that x has $m = 0.000001$ and $e = -95$ one needs to know that $emin = -95$ for this format.

13.4.2. *Exact representations of comparable types.* The exact text representation of a bare interval of any type should also be a valid exact representation in any wider (in the sense of §12.5.1) type, which when converted back produces the mathematically same interval.

That is, let type $\mathbb{T}'$ be wider than type $\mathbb{T}$. Let $\boldsymbol{x}$ be a $\mathbb{T}$-interval and let

$$\boldsymbol{s} = \mathbb{T}\text{-}\texttt{intervalToExact}(\boldsymbol{x}).$$

Then

$$\boldsymbol{x}' = \mathbb{T}'\text{-}\texttt{exactToInterval}(\boldsymbol{s})$$

should be defined and equal to $\mathbb{T}'$-`convertType`(x).

[*Note. If* $\mathbb{T}$ *and* $\mathbb{T}'$ *are 754-conforming types, this property holds automatically, because of the properties of* `textToInterval` *and the fact that* $\boldsymbol{s}$ *is an interval literal.*]

14. Level 3 description

14.1. Level 3 introduction. Level 3 is where Level 2 datums are represented, and operations on them described, in terms of more primitive entities and operations. How this is done is implementation-defined. Implementation may be by hardware, software, or a combination of the two.

Level 3 entities are here called *objects*; they represent Level 2 datums and may be referred to as *concrete*, while the datums are *abstract*. An implementation shall behave as if the relation between Levels 2 and 3 is as follows.

– The set of boolean values, the set of integer values, the set of strings §10.1, the set $\mathbb{D}$ of decorations §11.2, and the set of the states returned by `overlap` function §10.6.4, 12.13.4 are regarded as being the same at Level 3 as at Levels 1 and 2.
– The bare interval objects are organized into disjoint sets, *concrete bare interval types*, that are in one-to-one correspondence with the abstract bare interval types of Level 2. As at Level 2, a decorated interval is an ordered pair (bare interval, decoration), so this induces a one-to-one correspondence between the abstract and the concrete decorated interval types.
– The number objects are organized into disjoint sets, *concrete number formats*, that are in one-to-one correspondence with the abstract number formats of Level 2.

Thus intervals of a particular type exist in four forms: bare or decorated, and in either case abstract datums at Level 2 or concrete objects at Level 3. Similarly, numbers of a particular format exist in two forms, abstract or concrete.

In this document, the same name is normally used for an abstract type or format and its concrete counterpart. This convention, and the term "object", are not intended to constrain the names that an implementation gives to types or formats, nor the data structures it uses.
[*Example. The format* `decimal64` *might denote either the set of representations (in the sense of IEEE 754 §3.2) of decimal64 numbers, or the set of numbers thus represented. Then for instance, all representations in the cohort (IEEE 754 §3.5.1) of floating-point number* 0.1 *are different objects, but represent the same datum.*]

14.2. Representation. Individual datums of an abstract type or format are *represented* by individual objects of its concrete type or format. While the correspondence between abstract and concrete types or formats as a whole is one-to-one, that between datums and objects is not so. The property that defines a representation, for a given type or format, is:

> Each datum shall be represented by at least one object. Each object shall represent at most one datum. (30)

An object that represents a datum is called *valid*; one that does not is called *invalid*.

That is, representation is a *partial function* that is *onto* but usually *not one-to-one*, from the set of objects to the set of datums of a given type or format. The set of valid objects is the domain of this function.
[*Examples. Let* $\mathbb{F}$ *be a 754 format and let* $\mathbb{T}$ *the derived (bare) inf-sup type. Three possible representations are:*

– **inf-sup** *form. Any* $\mathbb{T}$*-interval* $\boldsymbol{x}$ *is represented at Level 3 by the object* $(\mathtt{inf}(x), \mathtt{sup}(x))$ *of two Level 2 numbers – members of* F. *All intervals have only one Level 3 representation because operations* `inf` *and* `sup` *are uniquely defined at Level 2 §12.12.8: interval* $[0, 0]$ *has representation* $(-0, +0)$, *interval* Empty *has representation* $(+\infty, -\infty)$.
– **inf-sup-nan** *form. The objects are defined to be pairs* (l, u) *where* l, u *are members of* $\mathbb{F}$. *A nonempty* $\mathbb{T}$*-interval* $\boldsymbol{x} = [\underline{x}, \overline{x}]$ *is represented by an object* (l, u) *such that the values of* l *and* u *are* $\underline{x}$ *and* $\overline{x}$, *and* Empty *is represented by* $(\mathrm{NaN}, \mathrm{NaN})$. *Its valid objects are* $(\mathrm{NaN}, \mathrm{NaN})$, *together with all* (l, u) *such that* l, u *are not* NaN *and* $l \le u$, $l < +\infty$, $u > -\infty$.
– **neginf-sup-nan** *form. This is as the previous, except that the value of* l *is* $-\underline{x}$. *Its valid objects are* $(\mathrm{NaN}, \mathrm{NaN})$, *together with all* (l, u) *such that* l, u *are not* NaN *and* $0 \le l + u$, $l > -\infty$, $u > -\infty$.

If, in these descriptions l, u *and* NaN *are viewed as Level 2 datums, then interval* $[0, 0]$ *has four representatives in* **inf-sup-nan** *and* **neginf-sup-nan** *forms:* $(-0, +0)$, $(-0, -0)$, $(+0, +0)$, $(+0, -0)$. *Each nonempty interval with nonzero bounds has only one representative: there are unique* l *and* u. Empty *has also only one representative: there is an unique* NaN. *However,* NaN *itself has representatives, and*

from this viewpoint Empty *has more than one representative: there are many* NaNs*, quiet or signaling and with different payloads, to use in* Empty = (NaN, NaN)*.*
]

14.3. Operations and representation. Each Level 2 (abstract) library operation is implemented by a corresponding Level 3 (concrete) operation, whose behavior shall be consistent with the abstract operation. That is, let $y = \varphi(x_1, x_2, \ldots)$ be a Level 2 operation instance whose inputs and output are any mix of number, interval, decoration, string or boolean datums, and let objects $x'_1, x'_2, \ldots$ represent $x_1, x_2, \ldots$, respectively. Then $y' = \varphi(x'_1, x'_2, \ldots)$ shall be defined and be a representative of y.

Since for each Level 2 operation, the result is defined for arbitrary input datums, it follows that each Level 3 library operation has a unique result, up to representation, for arbitrary *valid* input objects. That is, if one chooses different representatives x'_i for the x_i, the result y' may be different but is still a representative of y. The result, when some inputs are invalid, is implementation-defined. An implementation shall provide means for the user to specify that an exception `InvalidOperand` to be signaled when this occurs.

To promote reproducibility (§1.8, 6.5), an implementation should provide a computational mode where, provided certain programming restrictions are adhered to, the computed *values* depend only on the inputs.
[*Note. Such an effect might be achieved by canonicalizing, i.e., ensuring that, at least for operations with numeric output, the* representative *of the output is independent of the representatives of the inputs—in the notation above, y' is not changed by changing the x'_i. However, doing so incurs a cost, and there will be an implementation-defined trade-off between the run time overhead of a "reproducible mode", and its scope, i.e., how few programming restrictions it imposes.*

As an example, let $\mathbb{F}$ be a 754 decimal format and $\mathbb{T}$ 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

$$f(x) = \texttt{sameQuantum}(\texttt{inf}(\boldsymbol{x}), 0.3)$$

(see 754 §5.7.3). Suppose a program reads $\boldsymbol{x}$ using `textToInterval`*, first from the string* `[0.3,inf]`*, then from* `[0.30,inf]`*. A straightforward implementation might store them with the two Level 3 representations $\boldsymbol{x}' = (0.3, +\infty)$ and $\boldsymbol{x}'' = (0.30, +\infty)$, where 0.3 and 0.30 stand for the 754 decimal number objects $(0, -1, 3)$ and $(0, -2, 30)$ denoting the same real value $0.3 = 3 \times 10^{-1} = 30 \times 10^{-2}$. Thus $\boldsymbol{x}' = \boldsymbol{x}''$ at both Level 1 and Level 2, and the user might expect they give the same output, but*

$$f(\boldsymbol{x}') = \texttt{sameQuantum}(0.3, 0.3) \qquad = \texttt{true};$$
$$f(\boldsymbol{x}'') = \texttt{sameQuantum}(0.30, 0.3) \qquad = \texttt{false}.$$

The standard does not say which of these results is "correct". Rather than change the implementation, it might be better to document that such code must be avoided if reproducible behavior is required.]

14.4. Interchange representations. The purpose of interchange representations 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 representation using Level 2 number datums and delegating choice of interchange format of number datums to the IEEE 754 standard.

Let $\boldsymbol{x}$ be a datum of the bare interval inf-sup type $\mathbb{T}$ derived from a supported 754 format $\mathbb{F}$. Its standard Level 3 representative is an ordered pair $(\texttt{inf}(x), \texttt{sup}(x))$ of two Level 2 $\mathbb{F}$-numbers as defined in §12.12.8. For example, the only representative of Empty is the pair $(+\infty, -\infty)$ and the only representative of $[0, 0]$ is the pair $(-0, +0)$.

Let $\boldsymbol{x}_{dx}$ be a datum of the decorated interval type $\mathbb{DT}$ derived from $\mathbb{T}$. Its standard Level 3 representative is an ordered triple $(\texttt{inf}(\boldsymbol{x}), \texttt{sup}(\boldsymbol{x}), dx))$ of two Level 2 F-datums and a decoration. For example, the only representative of $\text{Empty}_{\texttt{trv}}$ is the triple $(+\infty, -\infty, \texttt{trv})$ and the only representative of NaI is the triple $(\text{NaN}, \text{NaN}, \texttt{ill})$.

Interchange representation of an interval datum comprises the interchange representations of fields of the ordered pair/triple above, in the order given. Interchange representation of the decoration field is an integer according with the table

Decoration	`ill`	`trv`	`def`	`dac`	`com`
Representation	0	1	2	3	4

Interchange representations of fields are not completely specified by this standard. Choices that are implementation-defined, and that an implementation shall document, include:

– choice of wider 754 interchange format if $\mathbb{F}$ is not 754 interchange format (754 §3.6) itself.
– choice of densely packed decimal (DPD) or binary integer decimal (BID) significand encoding for decimal 754 interchange formats.
– choice of size in bytes of integer decoration representation (one byte is recommended).

Issues of endianness, which affect how interchange representations map to sequences of bytes at Level 4, are outside the scope of the standard.

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

A 754-conforming implementation shall provide an interchange representation for each supported 754 interval type. Interchange representations for non-754 interval types, 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 representation is implementation-defined.

CHAPTER 3

Kaucher Intervals

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

ANNEX A

Including a new flavor in the standard

Additional flavors can be included in this standard. Standard-conforming flavors shall conform to the specifications in clauses 7 and 8. Official acceptance of a new flavor is done by submitting a Project Authorization Request (PAR) to the IEEE Standards Association for an amendment. The procedures for submitting an amendment shall be those as outlined in the IEEE Standards Style Manual, although the editing instructions normally would involve only a statement to insert an annex corresponding to the new flavor. The new flavor shall be vetted with the same care as the base standard. It is the responsibility of the interested parties to form the working group, submit the PAR, reach consensus, and see the amendment through the Sponsor Ballot.

ANNEX B

Set-based flavor: implementation hints and examples

B1. Type conversion

Interval type conversion (the hull operation) between the types of a 754-conforming implementation should be implemented in terms of the floating-point operations *formatOf*-`convertFormat` defined in 754 §5.4.2, with the appropriate outward rounding.

B2. Operation tables for basic interval operations

The tables in this subclause are an explicit realization of the general definition of interval operations in the set-based flavor given in §10.4. They are not normative, but are one possible basis for coding the interval versions of $+$, $-$, $*$, $/$. For fuller details see [?? suitable citation].

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 $\mathbb{F}$,
$\triangledown\!\!\!\!\circ$, etc.	:	the operations for elements of $\mathbb{F}$ with rounding downwards,
$\triangle\!\!\!\!\circ$, 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:

$$\boldsymbol{a} \circ \boldsymbol{b} := \{\, a \circ b \mid a \in \boldsymbol{a} \ \wedge\ b \in \boldsymbol{b} \ \wedge\ a \circ b \text{ is defined} \,\}, \tag{31}$$

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:

$$a \mathbin{\Diamond\!\!\!\!\circ}\, b := \ \Diamond\ (a \circ b), \tag{32}$$

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 $\Diamond\!\!\!\!\circ$, for $\circ \in \{+, -, *, /\}$, have the property

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

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

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

These operations and the unary operation $-\boldsymbol{a}$ can be expressed by more explicit formulas as shown in Tables B.2.1–B.2.4. 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	$[a_1, a_2] + [b_1, b_2] = [a_1 \triangledown b_1,\ a_2 \vartriangle b_2]$.
Subtraction	$[a_1, a_2] - [b_1, b_2] = [a_1 \triangledown b_2,\ a_2 \vartriangle b_1]$.

TABLE B.2.1. Negation, addition, subtraction.

Multiplication $[a_1, a_2] * [b_1, b_2]$	$[b_1, b_2]$ $b_2 \le 0$	$[b_1, b_2]$ $b_1 < 0 < b_2$	$[b_1, b_2]$ $b_1 \ge 0$
$[a_1, a_2], a_2 \le 0$	$[a_2 \triangledown b_2, a_1 \vartriangle b_1]$	$[a_1 \triangledown b_2, a_1 \vartriangle b_1]$	$[a_1 \triangledown b_2, a_2 \vartriangle b_1]$
$a_1 < 0 < a_2$	$[a_2 \triangledown b_1, a_1 \vartriangle b_1]$	$[\min(a_1 \triangledown b_2, a_2 \triangledown b_1),$ $\max(a_1 \vartriangle b_1, a_2 \vartriangle b_2)]$	$[a_1 \triangledown b_2, a_2 \vartriangle b_2]$
$[a_1, a_2], a_1 \ge 0$	$[a_2 \triangledown b_1, a_1 \vartriangle b_2]$	$[a_2 \triangledown b_1, a_2 \vartriangle b_2]$	$[a_1 \triangledown b_1, a_2 \vartriangle b_2]$

TABLE B.2.2. Multiplication.

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

TABLE B.2.3. 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 a bound 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, $0 \in \boldsymbol{b}$ $[a_1, a_2]/[b_1, b_2]$	$\boldsymbol{b} =$ $[0, 0]$	$[b_1, b_2]$ $b_1 < b_2 = 0$	$[b_1, b_2]$ $0 = b_1 < b_2$
$[a_1, a_2] = [0, 0]$	$\emptyset$	$[0, 0]$	$[0, 0]$
$a_1 < 0, a_2 \le 0$	$\emptyset$	$[a_2 \triangledown b_1, +\infty)$	$(-\infty, a_2 \vartriangle b_2]$
$[a_1, a_2], a_1 < 0 < a_2$	$\emptyset$	$(-\infty, +\infty)$	$(-\infty, +\infty)$
$0 \le a_1, 0 < a_2$	$\emptyset$	$(-\infty, a_1 \vartriangle b_1]$	$[a_1 \triangledown b_2, +\infty)$

TABLE B.2.4. 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 B.2.3, 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.

B3. Complete arithmetic, dot product function

For each supported 754-conforming type, derived from a format $\mathbb{F}$, the implementation should provide **complete arithmetic** for $\mathbb{F}$, as specified in Kulisch and Snyder [**5**].

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 should be provided, see [**5**] for details.

– `convert` converts from a complete format to a floating-point format, or vice versa, or from one complete format to another.

– `completeAdd` and `completeSub` add or subtract two complete or floating-point format operands, of which at least one is complete, giving a complete format result.

– `completeMulAccum` 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, giving a complete format result.

The result of all operations may be converted if necessary to a specified result format by application of the `convert` operation.

B4. Care needed with cancelMinus and cancelPlus

(informative)

In this standard these operations only apply to bounded intervals, since it seems hard to frame a specification for unbounded ones that translates unambiguously to the finite precision (Level 2) situation.

They also deviate from the Motion 12 specification, by being 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.

[*Example. For an example of numerical difficulties, 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* $\boldsymbol{z}_1 =$ `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* $\boldsymbol{z}_2 =$ `innerMinus`$(\boldsymbol{y}, \boldsymbol{x})$ *is not defined. However, one cannot discriminate these cases using naive 3 digit arithmetic. Comparing* `width`$(\boldsymbol{x})$ *with* `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.*]

B5. Local decorations of arithmetic operations

B5.1. Forward-mode elementary functions. For each of the required functions φ of §10.5, with the decoration scheme `com` $>$ `dac` $>$ `def` $>$ `trv` $>$ `ill` of Clause 11, Tables B.5.1 to B.5.2 give the **strongest local decoration** for arbitrary interval inputs. That is, they give $\mathrm{Dec}(\varphi \,|\, \boldsymbol{x})$ 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`, `abs`, `min`, `max`.

The functions φ in Table B.5.2 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 B.5.2 merely displays the set J for each function.

B6. 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 $\mathrm{Dec}(f \,|\, \boldsymbol{x}) =$ `ill` for all $\boldsymbol{x}$—though evaluation can never find this value, see examples below.
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 $\mathrm{Dec}(f \,|\, \boldsymbol{x}) =$ `dac` for any nonempty interval $\boldsymbol{x}$. But for any $\boldsymbol{x}$ of more than one point, evaluating $f(\boldsymbol{x})$ as in §11.6 gives `trv` as the decoration, because it takes the square root of an interval containing negative points.

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

   Note also that though `trv` is trivial in itself, to have $\mathrm{Dec}(f \,|\, \boldsymbol{x}) =$ `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 $\mathrm{Dom}(f)$, the second that $\boldsymbol{x}$ has a point outside $\mathrm{Dom}(f)$; 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 $\mathrm{Dom}(f)$ 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 $\mathrm{Dom}\, f$. Applying the `newDec` function gives initial decorated intervals $\boldsymbol{x}_{dx} = [1,2]_{\texttt{dac}}$, $\boldsymbol{y}_{dy} = [3,4]_{\texttt{dac}}$. The first operation is

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

TABLE B.5.1. Local decorations of required forward elementary functions. Normal mathematical notation is used to include or exclude an interval bound, e.g., $(-1,1]$ denotes $\{ x \in \mathbb{R} \mid -1 < x \le 1 \}$. It is assumed that interval types of arguments and result include bounded intervals large enough to contain modest numbers such as $\pm\pi$. 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 $\varphi(x_1, x_2, \ldots)$	$\begin{cases} \texttt{com} & \text{if inputs bounded, and result bounded at Level 2;} \\ \texttt{dac} & \text{otherwise.} \end{cases}$
$\texttt{div}(x,y)$	$\begin{cases} \texttt{com} & \text{if } 0 \notin \boldsymbol{y}\text{, inputs bounded, and result bounded at Level 2;} \\ \texttt{trv} & \text{if } 0 \in \boldsymbol{y}; \\ \texttt{dac} & \text{otherwise.} \end{cases}$
$\texttt{recip}(x)$	$\begin{cases} \texttt{com} & \text{if } 0 \notin \boldsymbol{x}, \boldsymbol{x} \text{ bounded, and result bounded at Level 2;} \\ \texttt{trv} & \text{if } 0 \in \boldsymbol{x}; \\ \texttt{dac} & \text{otherwise.} \end{cases}$
$\texttt{sqrt}(x)$	$\begin{cases} \texttt{trv} & \text{if } \boldsymbol{x} \not\subseteq [0,+\infty]; \\ \texttt{com} & \text{if } \boldsymbol{x} \subseteq [0,+\infty], \boldsymbol{x} \text{ bounded, and result bounded at Level 2;} \\ \texttt{dac} & \text{otherwise.} \end{cases}$
$\texttt{pown}(x,p),\ p \ge 0$	"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 $\mathcal{D} = \mathbb{R} \setminus \{\text{odd multiples of } \pi/2\}$.
$\texttt{asin}(x), \texttt{acos}(x)$	$\begin{cases} \texttt{com} & \text{if } \boldsymbol{x} \subseteq [-1,1]; \\ \texttt{trv} & \text{otherwise.} \end{cases}$
$\texttt{atan2}(y,x)$	$\begin{cases} \texttt{trv} & \text{if } \underline{y} \le 0 \le \overline{y} \text{ and } \underline{x} \le 0 \le \overline{x}; \\ \texttt{def} & \text{if } \underline{y} < 0 \le \overline{y} \text{ and } \overline{x} < 0; \\ \texttt{com} & \text{if } \boldsymbol{y} \text{ and } \boldsymbol{x} \text{ are bounded, and either } \overline{y} < 0 \text{ or } 0 < \underline{y} \text{ or } 0 < \underline{x}; \\ \texttt{dac} & \text{otherwise.} \end{cases}$ Note reversal of arguments y, x compared with mathematical definition x, y.
$\texttt{acosh}(x)$	$\begin{cases} \texttt{trv} & \text{if } \boldsymbol{x} \not\subseteq [1,+\infty]; \\ \texttt{com} & \text{if } \boldsymbol{x} \subseteq [1,+\infty], \boldsymbol{x} \text{ bounded, and result bounded at Level 2;} \\ \texttt{dac} & \text{otherwise.} \end{cases}$
$\texttt{atanh}(x)$	$\begin{cases} \texttt{trv} & \text{if } \boldsymbol{x} \not\subseteq (-1,1); \\ \texttt{com} & \text{if } \boldsymbol{x} \subseteq (-1,1)\text{, and result bounded at Level 2;} \\ \texttt{dac} & \text{otherwise.} \end{cases}$

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 $du' = \mathrm{Dec}(- \mid (\boldsymbol{y}, \boldsymbol{x})) = \texttt{dac}$, whence by (18) 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

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

TABLE B.5.2. 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.
$\mathtt{sign}(x)$	$J = \{0\}$
$\mathtt{ceil}(x), \mathtt{floor}(x)$	$J = \mathbb{Z}$
$\mathtt{trunc}(x)$	$J = \mathbb{Z} \setminus \{0\}$
$\mathtt{roundTiesToEven}(x), \mathtt{roundTiesToAway}(x)$	$J = \{ n + \frac{1}{2} \mid n \in \mathbb{Z} \}$

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

```
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
    $d = \mathtt{com}$
  else
    $d = \mathtt{dac}$
  end if
else
  $d = \mathtt{def}$
end if
Output: range enclosure $[\underline{y}, \overline{y}]$ and local decoration $d$.
```

The constant 1, following §10.4, becomes a decorated interval function returning the constant value $[1,1]_{\mathtt{dac}}$. The next operation is again "nice", and gives

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

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

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

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

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

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 $\mathrm{Dom}\, f$ so the true value of $\mathrm{Dec}(f \mid \boldsymbol{z})$ is still $\mathtt{dac}$. 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 $dw' = \mathrm{Dec}(\sqrt{\cdot} \mid \boldsymbol{w}) = \mathrm{Dec}(\sqrt{\cdot} \mid [-\frac{1}{2}, 5]) = \mathtt{trv}$ giving $\mathrm{Dec}(\sqrt{\cdot} \mid \boldsymbol{w}_{dw}) = \min\{dw', dv\} = \mathtt{trv}$, 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 $\mathtt{dac}$ property.

(iii) If $\boldsymbol{x} = [1,2]$, $\boldsymbol{y} = [1,1]$, the box $\boldsymbol{z}$ is now wholly outside $\mathrm{Dom}\, f$, 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 $\mathrm{Dom}\, f$, but owing to widening, evaluation fails to detect this, giving $\boldsymbol{f}_{df} = [0,0]_{\mathtt{trv}}$—a valid enclosure but of little use.

B7. Implementation of compressed interval arithmetic

Table B.7.1 gives tables of compressed arithmetic, §11.10, for the four basic operations. Here c, d are 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

$$\texttt{trv} \le c < \tau,\ \texttt{trv} \le d < \tau, \text{ and } \boldsymbol{x},\ \boldsymbol{y} \text{ are nonempty.} \tag{33}$$

TABLE B.7.1. Compressed interval operations for $+, -, \times, \div$ and $\sqrt{\cdot}$ with threshold $\tau \in \{\texttt{trv}, \texttt{def}, \texttt{dac}, \texttt{com}\}$.

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 (33) are assumed.

(a) Justification for `emp` $+ \boldsymbol{x} =$ `emp`, independent of τ.
This promotes to $(\emptyset, \texttt{emp}) + (\boldsymbol{x}, \tau) = \big(\emptyset, \min(\texttt{emp}, \tau, \texttt{emp})\big)$. Since `emp` $< \tau$ this equals $(\emptyset, \texttt{emp})$ which gives an exception (again because `emp` $< \tau$) so is recorded as the 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 $d \ge \texttt{trv}$, 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 $d \le e$, so $\min(\tau, d, e) = d$.

(c) Justification for $c/d =$ `trv` independent of τ.
Since $c, d \ge \texttt{trv}$, 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 $e =$ `emp` if $\boldsymbol{y} = [0, 0]$, else $e =$ `trv` if $0 \in \boldsymbol{y}$, else $e =$ `dac`. So $\min(c, d, e) \ge \texttt{trv}$ and can equal `trv`, so the tightest enclosing decoration is `trv`.

(d) Justification for $\boldsymbol{x}/\boldsymbol{y}$ when $0 \in \boldsymbol{y} \ne [0, 0]$.
$\boldsymbol{x}/\boldsymbol{y}$ promotes to $(\boldsymbol{x}, \tau)/(\boldsymbol{y}, \tau)$ giving $\big(\boldsymbol{x}/\boldsymbol{y}, \min(\tau, \tau, \texttt{trv})\big) = (\boldsymbol{x}/\boldsymbol{y}, \texttt{trv})$. If $\tau >$ `trv` this gives an exception so the decoration `trv` is returned; if $\tau =$ `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 $\underline{x} < 0 \le \overline{x}$.
$\sqrt{\boldsymbol{x}}$ promotes to $\sqrt{(\boldsymbol{x}, \tau)}$, giving $(\sqrt{\boldsymbol{x}}, \texttt{trv})$ which in the given case equals $([0, \sqrt{\overline{x}}], \texttt{trv})$. As with the previous item, if $\tau >$ `trv` then `trv` is returned; if $\tau =$ `trv` then $\sqrt{\boldsymbol{x}}$ is returned.

B8. The fundamental theorem of decorated interval arithmetic

This subclause states and proves the Fundamental Theorem 6.1 for the set-based flavor, in a computationally verifiable form where the decoration system is used to identify the different cases of that theorem. It thereby proves that a conforming implementation of this flavor can be used to conclude while evaluating a point function that it is everywhere defined, or everywhere defined and continuous, etc., on an input box.

B8.1. Preliminaries.

The variables in an expression. We denote the set of variables occurring in expression f by $V(\mathsf{f})$. This set is defined as follows:

– If f is a single named variable, e.g. x, then $V(\mathsf{f}) = \{\mathsf{x}\}$. The x is here regarded as a symbol, chosen from some set of allowed symbols.

– If $\mathsf{f} = \mathsf{h}(\mathsf{g}_1, \dots, \mathsf{g}_k)$ then

$$V(\mathsf{f}) = \bigcup_{i=1}^{k} V(\mathsf{g}_i).$$

In particular, if $k = 0$ (so f is a constant, see below) then $V(\mathsf{f}) = \emptyset$.

Argument association by keyword. It is helpful to use both positional and keyword notation for associating dummy arguments to actual arguments when invoking a function

Library operations φ use positional notation $\varphi(v_1, \dots, v_k)$ with the arguments in a fixed order, and the names of the dummy arguments are not important. However the theorem is simpler to state and prove if, for a function defined by expression, the variables occurring in the expression are linked *by name* to their actual arguments. E.g., expression $\mathsf{f} = b/d + a$ is deemed to define a function f with arguments a, b, d in no specified order. Keyword notation[1] is illustrated by $f(a = 5, b = 12, d = 4)$, which denotes the evaluation $12/4 + 5$ with result 8.

Formally, positional notation numbers the elements of $V(\mathsf{f})$ from 1 to n and indexes actual arguments x_i to match, making a map $x : i \mapsto x_i$ defined on $\{1, \dots, n\}$; while keyword notation indexes the arguments x_v directly by the variables v, making a map $x : v \mapsto x_v$ defined on $V(\mathsf{f})$.

Such an x is regarded as an actual-argument *vector*. When $U \subseteq V(\mathsf{f})$, notation x_U means the *subvector* comprising those x_v for $v \in U$, i.e., the restriction of map x to subset U of its domain.

Keyword notation helps solve two related problems, of *empty interval inputs* and of *ill-formed interval inputs*. These occur with positional notation when there is an argument that does not actually occur in the expression.

[*Example. Make the positional definition $y = f(a, b, c, d) = b/d + a$, where argument c does not occur in the defining expression $\mathsf{f} = b/d + a$.*

– *Suppose, with bare interval evaluation, we evaluate $\boldsymbol{y} = f(\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}, \boldsymbol{d})$ where $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{d}$ are nonempty but $\boldsymbol{c} = \emptyset$. What actually results from evaluating the expression f is $\boldsymbol{y}_C = \boldsymbol{b}/\boldsymbol{d} + \boldsymbol{a}$, which (whether at Level 1 or 2) is in general nonempty.*

  Using the positional definition, the input box is the product of 4 intervals, $\boldsymbol{x}_P = \boldsymbol{a} \times \boldsymbol{b} \times \boldsymbol{c} \times \boldsymbol{d} = \boldsymbol{a} \times \boldsymbol{b} \times \emptyset \times \boldsymbol{d} = \emptyset$, so that the theoretical range is $\boldsymbol{y}_P = \mathsf{Rge}(f \,|\, \emptyset) = \emptyset$.

  Using the keyword definition, the input box is the product of 3 intervals $\boldsymbol{x}_v$ for $v \in V(\mathsf{f})$, that is $\boldsymbol{x}_K = \boldsymbol{a} \times \boldsymbol{b} \times \boldsymbol{d} \neq \emptyset$, making the theoretical range to be $\boldsymbol{y}_K = \mathsf{Rge}(f \,|\, \boldsymbol{x}_K) \neq \emptyset$. In fact—if using the Level 1 interval versions of operations—$\boldsymbol{y}_C$ coincides with $\boldsymbol{y}_K$ whenever $0 \notin \boldsymbol{b}$.

– *A similar case arises with decorated interval evaluation, where inputs $\boldsymbol{a}_{da}, \boldsymbol{b}_{db}, \boldsymbol{d}_{dd}$ are well-formed but $\boldsymbol{c}_{dc}$ is NaI. Positional notation suggests the input box, and hence the result, should be NaI, but this result* cannot *come by evaluating the expression one operation at a time; the only way to get it is to redefine evaluation to include a preliminary scan that looks for NaI inputs.*

]

This example shows that keyword notation defines a theoretical result having a more useful and common-sense relation to the computed result of bare or decorated interval evaluation of an expression, than does positional notation.

Constant functions. It is necessary to clarify the case of a *zero-argument* arithmetic operation h. A point argument of a general k-ary h is a tuple $u = (u_1, \dots, u_k) \in \mathbb{R}^k$. For $k = 0$ this is the empty

[1]Similar to the keyword argument feature in Fortran, the `subs` command in Maple, etc.

tuple (), which is the unique element of $\mathbb{R}^0$. Since $\mathbb{R}^0$ is a singleton set, a particular $h : \mathbb{R}^0 \to \mathbb{R}$ is either total—defined everywhere—on $\mathbb{R}^0$, with a unique real value; or defined nowhere on $\mathbb{R}^0$.

– The total functions are in one to one correspondence with $\mathbb{R}$. If c is a symbol denoting a real number, the notation $c()$ means the total function on $\mathbb{R}^0$ with value c.
– The nowhere defined function on $\mathbb{R}^0$ is by definition the "Not a Number" function NaN.

In this way, each $h : \mathbb{R}^0 \to \mathbb{R}$ is either identified with a real constant—e.g., the functions $0.1()$ and $\pi()$ are identified with 0.1 and π—or is the unique NaN function.

The Level 1 interval version (the natural interval extension) of a total $c()$ is $\boldsymbol{c} : \mathbb{R}^0 \to \overline{\mathbb{IR}}$ with value $[c, c]$; that of the NaN function is the "Not an Interval" function NaI with value $\emptyset$.

The decorated version of $c()$ produces $[c, c]_{\texttt{com}}$; the decorated version of NaI produces $\emptyset_{\texttt{ill}}$.

At Level 2 one must consider finite precision. E.g., if $\mathbb{T}$ is the type inf-sup binary64, and $c = 0.1$, then $[c, c]$ is not a $\mathbb{T}$-interval and must be enclosed in a larger interval. A general interval extension of the constant $c()$ is any interval $[\underline{c}, \overline{c}]$ that contains c. Its decorated version is $[\underline{c}, \overline{c}]_{dc}$ where

$$dc = \begin{cases} \text{any decoration} \supseteq \texttt{com}, & \text{if } [\underline{c}, \overline{c}] \text{ is bounded,} \\ \text{any decoration} \supseteq \texttt{dac}, & \text{if } [\underline{c}, \overline{c}] \text{ is unbounded.} \end{cases}$$

Initialization of intervals. Define a decorated interval $\boldsymbol{X} = \boldsymbol{x}_{dx}$ to be **validly initialized** if either $\boldsymbol{x}_{dx} = \emptyset_{\texttt{ill}}$ or $p_{dx}(\text{Id}, \boldsymbol{x})$ holds, the latter being equivalent to $dx \supseteq \text{Dec}(\text{Id} \,|\, \boldsymbol{x})$. An $\boldsymbol{X}$ for which $dx \neq \texttt{ill}$ is called **tightly initialized** if $dx = \text{Dec}(\text{Id} \,|\, \boldsymbol{x})$. A decorated box is validly, or tightly, initialized if each of its decorated interval components is so.

The possible valid initializing decorations for the various kinds of interval are shown in the table below, with the tight initialization case underlined.

Interval $\boldsymbol{x}$	Valid initializing decoration
nonempty and bounded,	$\underline{\texttt{com}}$, `dac`, `def`, `trv`
unbounded,	$\underline{\texttt{dac}}$, `def`, `trv`
empty,	$\underline{\texttt{trv}}$
any nonempty bare interval; or $\underline{\text{empty}}$	`ill`

The tightly initialized cases coincide with those obtainable from a decorated constructor. If it succeeds, the decoration is as set by the `newDec` function. If it fails, the result is $\emptyset_{\texttt{ill}}$.

B8.2. The theorem.

Theorem B8.1 (Fundamental Theorem of Decorated Interval Arithmetic, FTDIA)**.** *Let* f *be an arithmetic expression denoting a real function f. Suppose* f *is evaluated, possibly in finite precision, on a validly initialized decorated box $\boldsymbol{X} = \boldsymbol{x}_{dx}$, with components indexed over V(f), to give result $\boldsymbol{Y} = \boldsymbol{y}_{dy}$.*

If some component of $\boldsymbol{X}$ is decorated `ill` *then*

$$dy = \texttt{ill} \tag{34}$$

.

If no component of $\boldsymbol{X}$ is decorated `ill`*, and none of the operations φ of* f *is an everywhere undefined function* Un*, then $dy \neq$* `ill` *and*

$$\boldsymbol{y} \supseteq \text{Rge}(f \,|\, \boldsymbol{x}) \quad \textit{and} \tag{35}$$

$$dy \supseteq \text{Dec}(f \,|\, \boldsymbol{x}). \tag{36}$$

In more primitive terms, (35, 36) say that $f(x) \in \boldsymbol{y}$ for all $x \in \boldsymbol{x}$, and $p_{dy}(f, \boldsymbol{x})$ holds.

Proof. We proceed by induction on the number of operations in f.

Base case. This is the case when f has zero operations. That is, it is a single variable, say x, and denotes the identity function $f = \text{Id} : x \mapsto x$ $(x \in \mathbb{R})$. Then the output $\boldsymbol{y}_{dy}$ equals the input decorated interval $\boldsymbol{X} = \boldsymbol{x}_{dx}$. [*Note. In finite precision it appears this need not be the case because of e.g., type conversion. However this is to be treated as a separate unary arithmetic operation—a decorated interval extension of* Id*—which must obey the usual rules for arithmetic operations.*]

Thus if $dx = \texttt{ill}$, then $dy = \texttt{ill}$ and (34) holds. If $dx \neq \texttt{ill}$, then $\text{Rge}(f \,|\, \boldsymbol{x}) = \text{Rge}(\text{Id} \,|\, \boldsymbol{x}) = \boldsymbol{x} = \boldsymbol{y}$ so (35) holds. Also $\texttt{ill} \neq dy = dx \supseteq \text{Dec}(\text{Id} \,|\, \boldsymbol{x}) = \text{Dec}(f \,|\, \boldsymbol{x})$ by the definition of valid initialization, so (36) holds. Hence the Theorem is true in this case.

Induction step. Suppose the Theorem has been established for all expressions with fewer than N operations, and let f have N operations.

By the definition of an expression, f is obtained from k expressions g_i, $i = 1, \dots, k$, through an arithmetic operation h of some arity $k \geq 0$. That is, $\mathsf{f} = \mathsf{h}(\mathsf{g}_1, \dots, \mathsf{g}_k)$, where necessarily each g_i has fewer than N operations. (Note the case $k = 0$, of a constant function, is included in this argument.)

Let $V_i = V(\mathsf{g}_i)$, that is, the set of variables occurring in g_i. Then $V(\mathsf{f}) = \bigcup_{i=1}^{k} V(\mathsf{g}_i)$. When f, h, and the g_i are regarded as defining point functions, this means $f(x) = h\big(g_1(x_{V_1}), \dots, g_k(x_{V_k})\big)$, where x_{V_i} is $V(\mathsf{g}_i)$ regarded as a vector.

By the inductive hypothesis the theorem holds for each g_i. That is, we have computed an interval $\boldsymbol{g}_i$ and a decoration dg_i, such that by (35, 36),

$$\boldsymbol{g}_i \supseteq \operatorname{Rge}(g_i \,|\, \boldsymbol{x}_{V_i}) \quad \text{and} \tag{37}$$

$$p_{dg_i}(g_i, \boldsymbol{x}_{V_i}) \quad \text{holds.} \tag{38}$$

By the definition of a decorated interval version of f, $\boldsymbol{y}_{dy}$ is computed using a decorated interval extension of h, hence by the definition in §11.6,

$$\boldsymbol{y} \supseteq \operatorname{Rge}(h \,|\, \boldsymbol{g}) \quad \text{and} \tag{39}$$

$$dy = \min\{\, dh, dg_1, \dots, dg_k \,\} \tag{40}$$

for some dh such that

$$p_{dh}(h, \boldsymbol{g}) \text{ holds.} \tag{41}$$

Corresponding to the different meanings of the decorations, (34, 35, 36) are verified case by case.

Case $df = \mathtt{ill}$. Then either some $dg_i = \mathtt{ill}$ or $dh = \mathtt{ill}$.
— If $dg_i = \mathtt{ill}$, by (38) $\operatorname{Dom} g_i$ is empty, and therefore $\operatorname{Dom} f$ is empty.
— If $dh = \mathtt{ill}$, then by (41) $\operatorname{Dom} h$ is empty, and therefore $\operatorname{Dom} f$ is empty.
In both cases, $\operatorname{Rge}(f \,|\, \boldsymbol{x}) = \emptyset$ and $p_{\mathtt{ill}}(f, \boldsymbol{x})$ holds. Since the computed result is $\emptyset_{\mathtt{ill}}$, (35, 36) hold.

Case $df = \mathtt{trv}$. This is always true, and nothing needs to be shown.

Case $df = \mathtt{def}$. Then each $dg_i \geq \mathtt{def}$. Hence, for all $i = 1, \dots, k$, $\boldsymbol{x}_{V_i}$ is a nonempty subset of $\operatorname{Dom} g_i$, and by (37), $\boldsymbol{g}_i \supseteq \operatorname{Rge}(g_i \,|\, \boldsymbol{x}_{V_i})$. Since also $dh \geq \mathtt{def}$, $\boldsymbol{g}$ is a nonempty subset of $\operatorname{Dom} h$. Hence f is everywhere defined on $\boldsymbol{x}$, that is $p_{\mathtt{def}}(f, \boldsymbol{x})$ holds, and (36) holds.

It remains to show (35). For any $v \in \operatorname{Rge}(f \,|\, \boldsymbol{x})$, there is $x \in \boldsymbol{x}$ such that $v = f(x)$. Then there exist $x_{V_i} \in \boldsymbol{x}_{V_i}$ for $i = 1, \dots, k$ such that

$$v = f(x) = h\big(g_1(x_{V_1}), \dots, g_k(x_{V_k})\big).$$

Denote $u_i = g_i(x_{V_i})$ and $u = (u_1, \dots, u_k)$. Since $u_i \in \operatorname{Rge}(g_i \,|\, \boldsymbol{x}_{V_i}) \subseteq \boldsymbol{g}_i$ and $u \in \boldsymbol{g}$, we have

$$\begin{aligned} v = f(x) &= h(u_1, \dots, u_k) \\ &\in \big\{h(u) \mid u \in \boldsymbol{g}\big\} \\ &= \operatorname{Rge}(h \,|\, \boldsymbol{g}) \subseteq \boldsymbol{y}. \end{aligned}$$

Since v was arbitrary, this proves (35). It holds in the next two cases since $\mathtt{dac}$ and $\mathtt{com}$ are stronger than $\mathtt{def}$.

Case $df = \mathtt{dac}$. This is as the $\mathtt{def}$ case with the addition that the restriction of each g_i to $\boldsymbol{x}_{V_i}$ is everywhere continuous, and the restriction of h to $\boldsymbol{g}$ is everywhere continuous. Hence the restriction of f to $\boldsymbol{x}$ is everywhere defined and continuous, and (36) holds.

Case $df = \mathtt{com}$. Then each $dg_i = \mathtt{com}$. Hence, for all $i = 1, \dots, k$, $\boldsymbol{x}_{V_i}$ is a bounded, nonempty subset of $\operatorname{Dom} g_i$, g_i is continuous at each point in $\boldsymbol{x}_{V_i}$, and the computed $\boldsymbol{g}_i$ is bounded. Since also $dh = \mathtt{com}$, h is defined and continuous at each point of $\boldsymbol{g}$, f is defined and continuous at each point of $\boldsymbol{x}$, and the computed $\boldsymbol{y}$ is bounded; (36) holds.

This completes the induction step and the proof. □

From the details of the proof one sees

Corollary B8.2. *If the result $\boldsymbol{y}_{dy}$ has $dy = \mathtt{ill}$, and the expression has at least one operation, then $\boldsymbol{y} = \emptyset$.*

This gives an argument in favor of assuming that if the decoration is `ill` then the interval part is *always* empty, i.e., of defining NaI to be $\emptyset_{\mathtt{ill}}$.

ANNEX C

Further material for set-based standard (informative)

C1. Specification of number literals within interval literals

This specifies the form and meaning of number literals in some common programming languages.

C/C++: A number literal is any valid input to the strtod function.

C2. Type conversion in mixed operations

⚠ This text is due to Arnold Neumaier, around 2010. It needs checking by language and compiler experts and should be the subject of a separate motion. It is not clear whether it belongs in this standard 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.

C3. The "Not an Interval" object

⚠ TO BE REVISED

From §6.1, 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 §10.4, 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] Griewank, Andreas *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation.* SIAM, Philadelphia, PA, (2000).

[3] Kulisch, Ulrich. Complete Interval Arithmetic and its Implementation on the Computer. Position paper, and the Dagstuhl 2008 proceedings.

[4] Kulisch, Ulrich. *Computer Arithmetic and Validity: Theory, Implementation, and Applications.* de Gruyter, Berlin, New York, (2008).

[5] 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.

[6] Moore, Ramon E. *Interval Analysis.* Prentice-Hall, Englewood Cliffs, N.J., (1966).

[7] Nehmeier, Marco and Siegel, Stefan and Wolff von Gudenberg, Jürgen. Specification of hardware for interval arithmetic. Computing 94, 243-255, (2012).

[8] Neumaier, Arnold. Vienna Proposal for Interval Standardization. Faculty of Mathematics, University of Vienna, (December 2008). `http://www.mat.univie.ac.at/~neum`

[9] Pryce, John D. and Corliss, George F. Interval arithmetic with containment sets. Computing 78, 251–276, (2006).

[10] Pryce, John D. P1788 Motion 6: Multi-Format Support: Text and Rationale, (2009).

[11] The Organization for the Advancement of Structured Information Standards (OASIS) Conformance Requirements for Specifications v1.0. `https://www.oasis-open.org/committees/ioc`