P1788/D8.0
Draft Standard For Interval Arithmetic

John Pryce and Christian Keil, Technical Editors

Frontmatter

Required IEEE items to be added:

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

⚠ To the P1788 working group reader.

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

Introduction

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

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

This document uses the standard notation $[a, b]$ for "the interval between numbers a and b", with various detailed meanings depending on the underlying theory. The "classical" interval arithmetic (IA) of R.A. Moore [**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, with a Kaucher flavor in preparation. Among other possible flavors are containment-sets, since they also extend classical IA in the defined sense, and systems that support open and half-open intervals.

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 may believe an FTIA holds in a case where it does not, with possibly serious effects in applications. As stated, the FTIA is about the mathematical level. Moore's achievements were to see that "outward rounding" makes the FTIA hold also in finite precision, and to follow through the consequences. An advantage of the level structure used by the standard is that the mapping between levels 1 and 2 defines a framework where it is easily proved that

The finite-precision FTIA holds in any conforming implementation.

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

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

Operations.

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

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

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

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

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

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

Decorations.

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

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

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

Contents

CHAPTER 1

General Requirements

1. Overview

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

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

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

1.3. Inclusions. This standard specifies

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

1.4. Exclusions. This standard does not specify

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

1.5. Word usage.

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

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

Further:

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

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

1.7. Programming environment considerations.

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

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

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

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

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

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

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

(1) define bindings so that the programmer can specify level 2 data (in the sense of the levels defined in 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, called *implementation* in the following, 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.

An implementation that conforms to this standard shall satisfy the following requirements:

- *flavor-independent requirements* [1]

A conforming implementation shall also provide at least one of the following profiles[2], defined in Clause 7, called included flavors of interval arithmetic in the context of this standard.

An implementation may also provide additional flavors that

- shall provide the required operations in Clause A.1;
- should provide the recommended operations in Clause A.2;
- shall provide common evaluations on common intervals as specified in §7.4.

[*Note. The procedure for submitting new flavors to be included into the standard is described in Annex B.*[3]]

⚠ The following paragraph will probably turn into a requirement on the library provided by an implementation, i.e. it has to provide a plain and a decorated library. For each provided flavor an implementation shall provide a decoration system supporting the verification of the assumptions of existence, uniqueness, or nonexistence theorems as specified in Clause 8. This includes as specified in §8.2

- a `newDec` function and
- a number of decorations; if the implementation provides more than one flavor, the decorations shall include the `com` decoration.

The normative version of this standard is the English version. Translation to other languages is permitted.[4]

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
- each supported bare interval type shall be based on a number format with an associated rounding function, see §12.4, and have 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, they shall shall be defined as a parameterized sequence of interval types, see §12.7
- evaluate expressions in a way that satisfies the equality principle, see §12.3,
- provide implementations of the required operations in §12.12; recommended accuracies for these operations can be found in §12.10
- if any of the recommended operations referenced in §12.13 are provided, they shall satisfy the requirements specified in the same clause
- 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 the afore mentioned clauses

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

[1] This is a placeholder and seems to stay empty for the moment—this will in turn lead to the paragraph being removed and requirements given for formats only.

[2] maybe drop this term?

[3] This is marked in the appendix as "has to be written", so is subject to change here as well.

[4] Maybe not the best place for this, but would need another subsection if placed after the flavors. In addition, what does the rest of the internationalization requirements in OASIS demand?

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, see §12.6. 754-conformance can be claimed for supported interval types if they satisfy the following requirements

- the supported interval type has to be 754-conforming and meet the requirements for mixed-type arithmetic, see §12.6, §12.12

3.2.2. *Compressed decorated interval arithmetic.* An implementation may support the compressed arithmetic subprofile of the set-based flavor. If compressed arithmetic is supported, it shall be as described in §11.11, 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 D.3 contains sample operation tables that satisfy this semantic
- provide implementations of the required operations in §12.12

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

- Kaucher requirements

3.4. Conformance claim. An implementation may claim its conformance to this standard in the following way[5]

> *Name of implementation and version* is conforming to the P1788 Standard for Interval Arithmetic, version D7.4[6] 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-included flavors*.

Part of the conformance claim shall be the completion of the following questionnaire[7].

3.5. Implementation conformance questionnaire.

(1) Implementation-defined behavior[8]
   (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.

(1) 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) how to convert the public representation of an interval to the represented interval datum as specified in §13.4

(2) Implementation-defined behavior
   (a) Does the implementation include the interval overlapping function, see §10.7.3? If so, how is it made available to the user?

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

[6] Update this version number.

[7] Is this normative or informative

[8] This is for now the only implementation-defined behavior outside of the set-based flavor. Things like recommended operations might be flavor independent, but are placed inside the set-based chapter for now. If we generalize these requirements or recommendations for other flavors, this section would expand.

(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, 12.4?
(c) What means if any does the implementation provide for an exception to be raised when a NaI is produced, see §11.3?
(d) What interval types are supported besides the required ones, see §12.1?
(e) How is the distance to an infinity calculated when rounding a number? How are ties broken in rounding numbers if multiple numbers qualify as the rounded result, see §12.4?
(f) Does the implementation include different versions of the same operation and how are these provided to the user, see §12.12?
(g) How is a failed constructor call signaled[9], see §12.12.8, 13.4?
(h) What combinations of formats are supported in interval constructors, see §12.12.8?
(i) What is the tightness of the result of constructor calls in cases where the standard does not require it, see §12.12.8?
(j) Does the implementation include compressed interval arithmetic and how is provided to the user [10], see §12.14?
(k) How are strings read from or written to streams[11], see §13.1?
(l) 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?
(m) How are the level 2 concepts represented and provided in terms of level 3 operations and objects[12], see §14.1?
(n) What is the result of level 3 operations for invalid inputs, see §14.3?
(o) Which cohort is used for a decimal number and are qNaNs or sNaNs used when converting an interval to the interchange format, see §14.5? What payload is embedded into the NaNs in this conversion?
(p) If the implementation provides non-standard decorations, what are these decorations and their mathematical definition, see §8.2? How are these decorations mapped when converting an interval to the interchange format, see §14.5?
(q) What interchange formats if any are provided for non-754 interval formats and on non-754 systems, see §14.5? How are these provided to the user?

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

(1) What decorations does the flavor provide and what is their mathematical definition?

3.6. Things to do and WIP. This subsection is just for keeping track of what needs to be done for this section.

3.6.1. *Things to consider.*

- a reproducibility mode might look like a level? at least it is a kind of quality of the implementation
  - This is currently described in Annex C with the first sentence "to be written" and has to be revisited when this changes, also defining whether it's a shall, a level, or a qoi.
- perhaps we have to clarify (here?) the difference between conformance levels and the levels structure—if necessary add something to the definitions.
  - As long as we don't have conformance levels, that's not necessary. If we have a basic standard, we introduce some language to clarify levels structure vs conformance level.
- the flavor text sounds like a flavor might be conforming to the standard? see §8.2
  - This is twofold. The one thing is the documentation of inlcuded flavors, see next point. the other thing is non-inluded flavors. This can be seen—and is above—as

[9] Is this just the kind of exception? Or also how it is signaled?
[10] is this the meaning of "how the operations are handled"? This may be moved to another place in the questionnaire as compressed arithmetic might move to be a kind of profile
[11] this seems obvious and should be removed from the text
[12] this should be more concrete—what is the implementation-defined part here?

part of the conformance of an implementation. But: Clause 7 says "a conforming implementation [...] may also provide non-included flavors, without losing conformance for the included flavors" and gives a set of "enforced behavior" for flavors. This seems contradictory as conformance may hold for included flavors but not for non-included ones while also saying a behavior is enforced. Also the concept of some flavors conforming and others not is complicating stuff. Better: conforming means all flavors have to conform, also non-included ones. Something like "A conforming implementation may provide non-included flavors. These non-included flavors shall satisfy the follwing common core of behavior."

- Set-based level 2 seems to contradict itself on the requirements of mixed-type arithmetic: §12.6 seems to require mixed-type operations for all level 2 operations, while §12.12 lists only a few explicitly.
- Why does set-based level 2 only reference the first 11 subsections of the required operations section in its header, see §12.12?
- The implementation-defined part of tie-braking in rounding seems to disagree between §12.4 and §12.12.9. The former one says tie-braking is implementation-defined subject to rounding direction, the latter one fixes it for certain formats.
- §10.4.2: "It is not specified how an implementation provides library facilities." Is this an implementation-defined item?
- We might want to claim conformance to the OASIS requirement itself.

3.6.2. *Things considered.*

- a language definition may be supporting the standard, an implementation may be conforming - may we have different objects that may be the object of the standard after all, see §1.7
  - A programming environment is conforming. this might consist of some combination of a language definition and extensions thereof.
- an implementation might also be a library, class or package
  - See previous point, hopefully clear with the introductory paragraph.
- compressed arithmetic certainly sounds like a profile—you may provide it, but if you do and want it to be conforming, you have to do it this way.
  - and it is a subprofile of set-based now.

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

- ✓ provide a conformance clause—we are doing this right now
- ✓ use the proper conformance keywords—as long as we are adhering to §1.5 this is covered.
- address all topics in section 8 of the OASIS requirements
  - ✓ (8.1) Specify what conforms—an implementation of an interval arithmetic programming environment
  - ✓ (8.1.1) Modularity of the conformance—We don't have different components that individually have to conform.
  - ✓ (8.1.2) Specifying conformance claims
  - ✓ (8.2) Profiles and levels
    - Flavors are profiles
    - We'll probably will have a limited set of requirements for a level of "basic" conformance
  - (8.3) Extensions—Allow or disallow? The `newDec` function sounds a bit like an extension, but it shows how difficult extensions might be with enhancements of functions that are not valid for all implementations. Additional decorations might be extensions as well. Maybe excluding extensions for flavors is a good idea. Besides that requiring that extensions must not cause non-conformant behavior.
  - ✓ (8.4) Discretionary items—implementation defined things
  - ✓ (8.5) Deprecated things—not yet
  - ✓ (8.6) Internationalization—English is normative, we have some characteristics for languages which might be interesting here[13]

[13] What is required besides specifying the normative version is unclear.

- examine and if appropriate address all things in section 9
  - ✓ (9.1) Implementation Conformance Statement—this might be relevant to register implementation defined points
  - ○ (9.2) Test assertions—probable connection to the test suite
  - ○ (9.3) Testing methodology/program

3.6.4. *Sections and references here.*

- Clause 6 No items here
- Clause 8 referenced with `newDec` and `com` ⚠ `newDec` now 'should' & not referenced by name.
- §10.1 – §10.5 probably only informational background
- §12.9 not referenced as only a definition

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.
$\mathbb{F}, \mathbb{G}, \ldots$	generic notation for the set of numbers, including $\pm\infty$, representable in some number format.
$\overline{\mathbb{IF}}, \overline{\mathbb{IG}}, \ldots$	the members of $\overline{\mathbb{IR}}$ whose lower and upper bounds are in $\mathbb{F}, \mathbb{G}, \ldots$.
Empty	the empty set.
Entire	the whole real line.
NaI	Not an Interval.
NaN	Not a Number.
qNaN, 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.

4.2. Definitions.

4.2.1. **754 format.** 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.** 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 in §12.6.

4.2.3. **754-conforming type.** 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.** 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.** A way to describe the quality of an interval version of a function. See §12.10.

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

Together with the interval non-arithmetic operations (§10.4.1), these form the implementation's **library**, which splits into the **point library** (a conceptual entity, being a set of mathematical functions), the **bare interval library** and the **decorated interval library**, corresponding to the above categories. The latter two may be further qualified by a result interval type, e.g., "binary64 inf-sup decorated interval library".

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

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

Constants such as 3.456 and π are regarded as arithmetic operations whose number of arguments is zero. Details in §10.4.4.

4.2.7. **available.** To be defined, see Motion 16 . . . [14]

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

4.2.9. **box.** A **box** or **interval vector** is an n-dimensional interval, i. e. a tuple $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$ where the $\boldsymbol{x}_i$ are intervals. Often identified with the cartesian product $\boldsymbol{x}_1 \times \ldots \times \boldsymbol{x}_n \subseteq \mathbb{R}^n$. In this sense, it is empty if any of the $\boldsymbol{x}_i$ is empty. Details in §10.2.

4.2.10. **comparable.** 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.

4.2.11. **conforming part.** 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 can be formed from a subset of the flavors. See §3.1.

4.2.12. **datum.** One of the entities manipulated by finite precision (Level 2) operations of this standard. It can be a **boolean**, a **decoration**, an **interval** (also called 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 §12.1.

4.2.13. **decoration.** One of the flavor-defined set of values used in the exception handling system. The set-based flavor uses the five values `com`, `dac`, `def`, `trv` and `ill`, see Clause 11.

4.2.14. **decorated interval.** A pair (interval, decoration).

4.2.15. **decorated interval library.** See Defn 4.2.6.

4.2.16. **decorated interval version.** See Defn 4.2.6.

4.2.17. **domain.** For a function with arguments taken from some set, the **domain** comprises those points in the set at which the function has a value. The domain of an arithmetic operation is part of its definition. E.g., the (point) arithmetic operation of division x/y, in this standard, has arguments (x, y) in $\mathbb{R}^2$, and its domain is the set of points in $\mathbb{R}^2$ where $y \neq 0$. See also Defn 4.2.35.

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

4.2.19. **expression.** A symbolic form used to define a function. The standard does not define the syntax or semantics of expressions, which are language- or implementation-defined. See Clause 6.

4.2.20. **explicit type.** An interval type that has a uniquely defined interval hull operation.

4.2.21. **floating-point format.** 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 b is the fixed radix, p 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 $emin \leq e \leq emax$.

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

4.2.23. **format.** (Or **number format**.) One of the sets into which number datums are organized at Level 2, usually regarded as a finite set of real numbers together with the values $\pm\infty$ and NaN. However the definition is such that members of different formats are not the same even when they have the same Level 1 value. Details in §12.4.

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 $\mathbb{F}$.

4.2.24. **hull.** (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 $\boldsymbol{s}$.

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.

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

4.2.26. **implicit type.** 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.

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

[14] *CK* 2011-11-08 CK

4.2.28. **interval.** (Bare interval.) A closed connected subset of $\mathbb{R}$; may be empty, bounded or unbounded. The set of all intervals is denoted $\overline{\mathbb{IR}}$. For **interval vector** see **box**.

At Level 2 a $\mathbb{T}$-interval is a member of the interval type $\mathbb{T}$, or a non-NaI member of the decorated interval type $\mathbb{T}$.

4.2.29. **interval extension.** 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 §10.4.3. 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.

4.2.30. **interval library.** See Defn 4.2.6.

4.2.31. **mathematical interval of constructor.** 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 $\boldsymbol{x}$. Details in §12.12.8.

4.2.32. **mid-rad.** Describes a representation of an interval based on its midpoint and radius.

4.2.33. **NaI, NaN.** NaN is the Not a Number datum, which is a member of every number format of this standard. NaI is the Not an Interval datum, which is a member of every decorated interval type of this standard.

4.2.34. **narrower, wider.** 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.

4.2.35. **natural domain.** For an arithmetic expression $f(z_1, \ldots, z_n)$, the natural domain is the set of $x = (x_1, \ldots, x_n) \in \mathbb{R}^n$ where the expression defines a value for the associated point function $f(x)$. See Clause 6.

4.2.36. **no value.** 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 $y = 0$.

4.2.37. **non-arithmetic operation.** An operation on intervals that is not an interval extension of a point operation; includes intersection and convex hull of two intervals.

4.2.38. **number.** Any member of the set $\mathbb{R} \cup \{-\infty, +\infty\}$ of extended reals: a **finite number** if it belongs to $\mathbb{R}$, else an **infinite number**. See §10.1.

4.2.39. **number format.** See **format**.

4.2.40. **operation.** Essentially synonymous with *function* and *mapping*.

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

4.2.42. **point library.** See Defn 4.2.6.

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

4.2.44. **string, text.** 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.

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

4.2.46. **tightness.** The strongest of the accuracy modes *tightest*, *accurate*, *valid*, that a given operation, in a given type, achieves for some input box, or uniformly over some set of inputs. See §12.10.

4.2.47. **type.** (Also **interval type**.) One of the sets into which bare and decorated interval datums are organized at Level 2, usually regarded as a finite set of Level 1 intervals, in the bare case; and of Level 1 decorated intervals together with the value NaI, in the decorated case. However the definition is such that members of different types are not the same even when they have the same Level 1 value.

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 $\mathbb{T}$-datum, but for decorated types means a non-NaI member of $\mathbb{T}$.

4.2.48. **version of an operation.** A finite precision approximation to a Level 1 operation. Typically but not necessarily, each interval input or output becomes one of some bare or decorated interval type $\mathbb{T}$. Details in §12.9.

4.2.49. **wider.** See **narrower**.

Relationships between specification levels for interval arithmetic for a given flavor $\mathfrak{F}$ and a given finite-precision bare interval type $\mathbb{T}$.		
Level 1	Number system used by flavor $\mathfrak{F}$. Set of allowed intervals in $\mathfrak{F}$. Principles of how $+$, $-$, $\times$, $\div$ and other arithmetic operations are extended to intervals.	Mathematical Model
	$\downarrow \mathbb{T}$-*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 $\mathbb{T}$-intervals, 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 exception handling mechanism.

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*[15]

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, may 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 may have more than one.

[15]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 may 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. Expressions and the functions they define

6.1. Definitions. An expression is some symbolic form used to define a function. Expressions are central to interval computation, because the Fundamental Theorem of Interval Arithmetic (FTIA) is about interpreting an expression in different ways:

- as defining a mathematical real point function f;
- 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 $\boldsymbol{x}$;
- as defining corresponding decorated interval functions that can give the stronger conclusion that f is everywhere defined, or everywhere continuous, on $\boldsymbol{x}$—enabling, for example, an automatic check of the hypotheses of the Brouwer Fixed Point Theorem.

The standard specifies behavior, at the individual operation level, that enables such conclusions, whether or not the notion "expression" exists in a host 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 graph, see below) set of dependences between mathematical variables, defined by equations

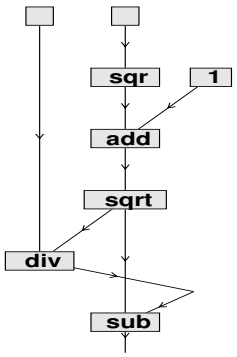
$$v = \varphi(u_1, \ldots, u_k), \tag{1}$$

where v and the u_i come from a nonempty finite set $\mathcal{X}$ of *variable-symbols*; φ comes from a finite set $\mathcal{F}$ of *formal library operations*; $k \geq 0$ is the *arity* of φ; and distinct equations have distinct v's—the *single assignment* property.

Three essentially equivalent descriptions of an expression are as follows.

(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.

To apply the FTIA it suffices to consider expressions that are *scalar*, with a single output. (All the individual library arithmetic operations of the standard are scalar.)



Computational graph
(a)

`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` $y = v_5$

Code list
(b)

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

Algebraic expression
(c)

FIGURE 6.1. Essentially equivalent notations for an expression. In (a), the 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. Note form (c) has redundancy; (a) and (b) do not.

(b) Since $\mathcal{G}$ is acyclic the equations can be ordered so that each one only depends on inputs 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 $v_{1-n}, \ldots, v_0$ where $n \geq 0$, conventionally given the aliases $x_1, \ldots, x_n$, so x_i is the same as v_{i-n}. The operations are

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

where $\varphi_r \in \mathcal{F}$ with arity 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.) Assuming a single output, it is v_m, given the alias y, so that the expression defines a formal 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*. If $m = 0$ and $n = 1$, there are no operations and y is the same as x_1, defining the *identity function* $y = f(x_1) = x_1$. In general for $m = 0$ and $n \geq 1$ there are n possibilities, the *coordinate projections* $\pi_j(x_1, \ldots, x_n) = x_j$ $(j = 1, \ldots, n)$.

(c) By allowing redundancy, an expression always can be converted to a normal *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 expression;
- if $f \in \mathcal{F}$ is a function symbol of arity k and if e_i is an expression for $i = 1, \ldots, k$ then the *function symbol application* $f(e_1, \ldots, e_k)$ is an expression.

The redundancy is because this form has no way of referring to intermediate values by name, so that if such a value is used several times, the subexpression that computes it must be repeated each time it occurs, see Figure 6.1. If the algebraic expression is evaluated naively, such a subexpression is evaluated more than once, which affects efficiency but not the numerics of what is computed.

Because of its simplicity, this is the definition of expression used in the FTIA proof in Clause D.4.

6.2. Mapping to a library. In IA applications, $\mathcal{F}$ and the arity of each of its functions are defined from the interface of an IA library. Formal expressions typically map to expressions in languages using IA libraries, or to sections of run-time data-flow in executions of programs using IA libraries. The statement of the FTIA transfers to applications of IA libraries via appropriate mapping of a formal expression to a language expression or a section of data-flow. In the absence of such a mapping, the conclusions of the FTIA cannot be drawn from execution of programs written in a host language.

Namely, each formal operation in $\mathcal{F}$ must have a primary *point version* with real-number input(s) and output; one or more *interval versions* with interval input and output; and for each of these a corresponding *decorated interval version*. These produce *point evaluation*, *interval evaluation* and *decorated interval evaluation* of the expression, also termed evaluation in *point mode*, *interval mode* or *decorated interval mode*. Floating-point versions are not relevant to this standard.

The point version is a theoretical (Level 1) function, of which each interval version—there is at least one for each interval type provided by the implementation—is a finite-precision (Level 2) *interval extension*, and each decorated interval version is a *decorated interval extension*.

An implementation's library by definition comprises all its Level 2 versions of required or recommended operations that it provides for any of its supported interval types. For the set-based flavor these are specified in §10.6, §10.7 and in Clause 12. Different interval evaluations of f come from using library operations of different Level 2 types, as the implementation may provide.

The set operations `intersection` and `convexHull` are not point-operations and cannot appear directly in an arithmetic expression. However they are useful for efficiently *implementing* interval extensions of functions defined piecewise, see Example (ii) in §11.9.

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

In the set-based flavor, Moore's basic theorem for a scalar function is as follows, with the above notation.

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

(i) ("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:

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

(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 $\boldsymbol{x}$.

It is important to note that the theorem holds in finite precision, not just at Level 1. The decoration system gives basic tools for checking the conditions for the "defined" and "continuous" forms, during evaluation of a function.

6.4. 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 the example in §11.9 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.

The standard requires that at Level 2, for all interval types, operations and inputs, 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)—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.

7. Flavors

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

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

A flavor is identified by a unique name. Certain flavors, termed **included**, are specified in this standard. The *(list to be confirmed)* flavors are the currently included flavors. The procedure for submitting a new flavor for inclusion is described in Annex B. An implementation that has both included and non-included flavors is not conforming as a whole, but the part (§3.1) comprising the included flavors may be conforming.

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

(i) There is a set of *common intervals* whose members are—in a sense made precise in §7.4—intervals of any flavor.

(ii) There is a set of library operations, identified by their names, that are required in all flavors; see Clause 9.

(iii) There is a set of *common evaluations* of library operations, with common intervals as input, that give—in a sense made precise in §7.2—the same result in any flavor.

In item (iii) the result means the tightest mathematical (Level 1) result, ignoring any interval widening due to finite precision (Level 2).

7.2. Definition of common intervals and common evaluations. This subclause specifies the set of common intervals, and the rules that determine the common evaluations for each of the required and recommended operations of the standard listed in Clause 9.

The common intervals comprise the set $\mathbb{IR}$ of *closed bounded nonempty real intervals* used in classical Moore arithmetic [**6**].

Each operation may be written $y = \varphi(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_k)$ where the formal arguments $\boldsymbol{x}_i$ and result $\boldsymbol{y}$ may be of interval type or of various other types such as real or boolean, but at least one is of interval type. The rules define those k-tuples $\boldsymbol{x} = (\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_k)$ for which $\varphi(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_k)$ shall have the same value $\boldsymbol{y}$ in all flavors.Then $\boldsymbol{x}$ is called a *common input* and $\boldsymbol{y}$ is the *common value* of φ at $\boldsymbol{x}$. The $(k+1)$-tuple $(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_k;\ \boldsymbol{y})$ is a *common evaluation*; it may be called a common evaluation *instance* to emphasize that a specific tuple of inputs is involved.

Two cases are distinguished:

– φ is an interval *arithmetic operation*, which is an interval extension (Defn 4.2.29), of the corresponding point function $\widehat{\varphi}$. Its inputs $\boldsymbol{x}_i$ are all intervals so that $\boldsymbol{x} = (\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_k)$ is a box, regarded as a subset of $\mathbb{R}^k$.

  The common inputs shall comprise those $\boldsymbol{x} = (\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_k)$ such that each $\boldsymbol{x}_i$ is common, and $\widehat{\varphi}$ is defined and continuous at each point of $\boldsymbol{x}$. The common value $\boldsymbol{y}$ shall be the range

$$\boldsymbol{y} = \mathrm{Rge}(\widehat{\varphi} \,|\, \boldsymbol{x}) = \{\, \widehat{\varphi}(x_1, \ldots, x_n) \mid x_i \in \boldsymbol{x}_i \text{ for each } i \,\}.$$

  Thus the common evaluations comprise a restriction, to a subset of the set of all possible boxes, of the *natural* interval extension of $\widehat{\varphi}$. Necessarily, by theorems of real analysis, $\boldsymbol{y}$ is nonempty, closed, bounded and connected, so it is a common interval.

– In all other cases, φ has a direct definition unrelated to a point function. The common inputs shall comprise those $\boldsymbol{x} = (\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_k)$ such that each $\boldsymbol{x}_i$ is common, and $\boldsymbol{y} = \varphi(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_k)$ is (a) defined and (b) if of interval type, is a common interval. The common value shall be $\boldsymbol{y}$.

[*Examples.*

1. *For interval division $\boldsymbol{x}_1/\boldsymbol{x}_2$ (an arithmetic operation), the common inputs are the $(\boldsymbol{x}_1, \boldsymbol{x}_2)$ with $\boldsymbol{x}_i \in \mathbb{IR}$ and $0 \notin \boldsymbol{x}_2$.*
2. *For interval square root $\sqrt{\boldsymbol{x}}$ (an arithmetic operation), the common inputs are the $\boldsymbol{x} = [\underline{x}, \overline{x}] \in \mathbb{IR}$ for which $0 \le \underline{x} \le \overline{x} < +\infty$.*
3. *For $\mathtt{intersection}(\boldsymbol{x}_1, \boldsymbol{x}_2) = \boldsymbol{x}_1 \cap \boldsymbol{x}_2$ (an interval-valued nonarithmetic operation), with inputs $(\boldsymbol{x}_1, \boldsymbol{x}_2)$ that are common intervals, the result is a common interval iff it is nonempty. Hence the common inputs are the $(\boldsymbol{x}_1, \boldsymbol{x}_2)$ with $\boldsymbol{x}_i \in \mathbb{IR}$ and $\boldsymbol{x}_1 \cap \boldsymbol{x}_2 \neq \emptyset$. For $\mathtt{convexHull}(\boldsymbol{x}_1, \boldsymbol{x}_2) = \mathsf{hull}(\boldsymbol{x}_1 \cup \boldsymbol{x}_2)$, there are no exceptions: all $(\boldsymbol{x}_1, \boldsymbol{x}_2)$ with $\boldsymbol{x}_i \in \mathbb{IR}$ are common inputs.*
4. *The midpoint function $\mathtt{mid}(\boldsymbol{x})$ (a real-valued nonarithmetic operation) has the formula $\mathtt{mid}(\boldsymbol{x}) = (\underline{x} + \overline{x})/2$ for every common interval $\boldsymbol{x} = [\underline{x}, \overline{x}]$. Thus every common interval is a common input.*
5. *The above definition for arithmetic operations requires R1 "$\widehat{\varphi}$ is defined and continuous at each point of $\boldsymbol{x}$", which is a* stronger *constraint (results in fewer common evaluations) than R2 "the restriction of $\widehat{\varphi}$ to $\boldsymbol{x}$ is everywhere defined and continuous". R1 produces a* weaker *constraint on what interval arithmetics can be flavors (with fewer rules, more arithmetics obey them). In particular, requirement R1 permits cset arithmetic to be a flavor, while R2 does not.*

   E.g., the common inputs for (the interval extension of) $\mathsf{floor}(x)$ *are all nonempty intervals that are disjoint from $\mathbb{Z}$. Thus* $\mathsf{floor}([1, 1.9]) = [1, 1]$ *is not common, because* $\mathsf{floor}()$ *is not continuous at 1, despite its restriction to $[1, 1.9]$ being everywhere continuous. If it were required to be common, cset arithmetic could not be a flavor.*

]

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

$$(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_k;\ \boldsymbol{y}') \qquad \text{with } \boldsymbol{y}' \in \mathbb{IR},\ \boldsymbol{y}' \supseteq \boldsymbol{y}. \tag{3}$$

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

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

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

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* $\boldsymbol{x} \in \mathbb{IR}$.]

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

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

$$\begin{array}{l}\text{If } (\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_k;\ \boldsymbol{y}) \text{ is a common evaluation instance of } \varphi,\\ \text{then } (\mathfrak{f}\boldsymbol{x}_1, \mathfrak{f}\boldsymbol{x}_2, \ldots, \mathfrak{f}\boldsymbol{x}_k;\ \mathfrak{f}\boldsymbol{y}) \text{ is an evaluation instance of } \varphi \text{ in flavor } \mathfrak{F}.\end{array} \tag{4}$$

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

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

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

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

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

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

8. Decoration system

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

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

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

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

For instance, if a section of code defines the expression $\sqrt{y^2 - 1} + xy$, then decorated-interval evaluation of this code with suitably initialized input intervals $\boldsymbol{x}, \boldsymbol{y}$ gives information about the

definedness, continuity, etc. of the point function $f(x, y) = \sqrt{y^2 - 1} + xy$ over the box $(\boldsymbol{x}, \boldsymbol{y})$ in the plane.

The decoration system is designed in a way that naive users of interval arithmetic do not notice anything about decorations, unless they inquire explicitly about their values. 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 may even relieve users from these tasks. Expert users can inspect, set and modify decorations to improve code efficiency, but are responsible for checking that computations done in this way remain rigorously valid.

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

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

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

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

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

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

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

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

The local decoration d may convey purely Level 1 information—e.g., that φ is everywhere continuous on the box $\boldsymbol{x} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k)$. It may convey Level 2 information related to the particular finite-precision interval types being used—e.g., that $\boldsymbol{y}$, though mathematically a bounded interval,

became unbounded by overflow. For diagnostic use it may convey Level 3 or 4 information, e.g., how an interval is represented, or how memory is used.

If f is an expression, decorated interval evaluation of an expression means evaluation of f with decorated interval inputs and using decorated interval extensions of the expression's library operations. Those inputs generally need to be given 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.

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

> Let φ applied to input intervals $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_k$ give the computed result $\boldsymbol{y}$, and let $\varphi(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_k) = \boldsymbol{y}$ be a loose common evaluation as defined in (3); in particular $\boldsymbol{y}$ is bounded. If each of the inputs $\boldsymbol{x}_i$ is decorated `com`, then the output $\boldsymbol{y}$ shall be decorated `com`.

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

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

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

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

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

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

]

9. Operations required in all flavors

This clause defines the required library arithmetic and non-arithmetic operations of the standard. It gives for each arithmetic operation the mathematical formula for the point function, its domain of definition, and the set where it is continuous if different from the domain. By §7.2, this data defines the common evaluations of the Level 1 interval version of each such operation.

An implementation shall provide each required operation in each included flavor. The behavior of the operation outside the set of common evaluations is flavor-defined.

9.1. Arithmetic operations.

Table 9.1 on page 21 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 $x \bullet y$.

Notes to Table 9.1

a. In describing the domain, notation such as $\{y = 0\}$ is short for $\{\, (x, y) \in \mathbb{R}^2 \mid y = 0 \,\}$, etc.

b. Regarded as a family of functions parameterized by the integer argument p.

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 endpoint, e.g., $(-\pi, \pi]$ denotes $\{\, x \in \mathbb{R} \mid -\pi < x \leq \pi \,\}$.

Name	Definition	Point function domain	Point function range	Notes
`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)	$(x \times y) + z$	$\mathbb{R}^3$	$\mathbb{R}$	
`pown`(x,p)	x^p, $p \in \mathbb{Z}$	$\begin{cases}\mathbb{R} \text{ if } p \geq 0 \\ \mathbb{R}\setminus\{0\} \text{ if } p < 0\end{cases}$	$\begin{cases}\mathbb{R} \text{ if } p > 0 \text{ odd} \\ [0,\infty) \text{ if } p > 0 \text{ even} \\ \{1\} \text{ if } p = 0 \\ \mathbb{R}\setminus\{0\} \text{ if } p < 0 \text{ odd} \\ (0,\infty) \text{ if } p < 0 \text{ even}\end{cases}$	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
`sin`(x)		$\mathbb{R}$	$[-1,1]$	
`cos`(x)		$\mathbb{R}$	$[-1,1]$	
`tan`(x)		$\mathbb{R}\setminus\{(k+\frac{1}{2})\pi \mid k \in \mathbb{Z}\}$	$\mathbb{R}$	
`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)		$(-1,1)$	$\mathbb{R}$	
`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
`abs`(x)	$\lvert x \rvert$	$\mathbb{R}$	$[0,\infty)$	
`min`$(x_1,\ldots,x_k)$		$\mathbb{R}^k$ for $k = 2,3,\ldots$	$\mathbb{R}$	k
`max`$(x_1,\ldots,x_k)$		$\mathbb{R}^k$ for $k = 2,3,\ldots$	$\mathbb{R}$	k

c. Defined as $e^{y \ln x}$ for real $x > 0$ and all real y, and 0 for $x = 0$ and $y > 0$, else undefined. 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 just outside the mathematical range.
f. 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. `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. `ceil`(x) is the smallest integer $\geq x$. `floor`(x) is the largest integer $\leq x$. `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. `roundTiesToEven`(x), `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. A family of functions parameterized by the arity k.

9.2. Cancellative addition and subtraction.

For common intervals $\boldsymbol{x} = [\underline{x}, \overline{x}]$, $\boldsymbol{y} = [\underline{y}, \overline{y}]$, the operation `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 `cancelPlus`$(\boldsymbol{x}, \boldsymbol{y})$ is equivalent to `cancelMinus`$(\boldsymbol{x}, -\boldsymbol{y})$.

9.3. Set operations.

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

– `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.
– `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. Numeric functions of intervals.

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

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`$(\boldsymbol{x})$	$\inf\{\, \lvert x\rvert \mid x \in \boldsymbol{x} \,\} = \begin{cases} \min(\lvert\underline{x}\rvert, \lvert\overline{x}\rvert) & \text{if } \underline{x}, \overline{x} \text{ have the same sign} \\ 0 & \text{otherwise} \end{cases}$

9.5. Boolean functions of intervals.

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

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`$(\boldsymbol{a}, \boldsymbol{b})$	$\boldsymbol{a} \subseteq \boldsymbol{b}$	$\forall_a \exists_b\, a = b$	$\underline{b} \leq \underline{a} \wedge \overline{a} \leq \overline{b}$	$\boldsymbol{a}$ is a subset of $\boldsymbol{b}$
`interior`$(\boldsymbol{a}, \boldsymbol{b})$	$\boldsymbol{a} \circledcirc \boldsymbol{b}$	$\forall_a \exists_b\, a < b \wedge \forall_a \exists_b\, b < a$	$\underline{b} < \underline{a} \wedge \overline{a} < \overline{b}$	$\boldsymbol{a}$ is interior to $\boldsymbol{b}$
`disjoint`$(\boldsymbol{a}, \boldsymbol{b})$	$\boldsymbol{a} \between \boldsymbol{b}$	$\forall_a \forall_b\, a \neq b$	$\overline{a} < \underline{b} \vee \overline{b} < \underline{a}$	$\boldsymbol{a}$ and $\boldsymbol{b}$ are disjoint

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 §10.5 describe the theory of mathematical intervals and interval functions that underlies this flavor. The relation between expressions and the point or interval functions that they define is specified, since it is central to the Fundamental Theorem of Interval Arithmetic. Subclauses §10.6, 10.7 list the required and recommended *arithmetic operations* (also called elementary functions) with their mathematical specifications. Clause 11 describes, at a mathematical level, the system of *decorations* that is used among other things for exception handling in this flavor of the standard.

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

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

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

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

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

$$[\underline{x}, \overline{x}] = \{\, x \in \mathbb{R} \mid \underline{x} \le x \le \overline{x} \,\}, \tag{5}$$

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

[*Notes.*

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

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

– *Mathematical literature generally uses a round bracket, or reversed square bracket, to show that an endpoint is excluded from an interval, e.g. $(a, b]$ and $]a, b]$ denote $\{\, x \mid a < x \le b \,\}$. Where it is convenient to change to this notation, this is pointed out, e.g., in the tables of function domains and ranges in §10.6, 10.7.*

– *The set of intervals $\overline{\mathbb{IR}}$ could be described more concisely as comprising all sets $\{\, x \in \mathbb{R} \mid \underline{x} \le x \le \overline{x} \,\}$ for* arbitrary *extended-real $\underline{x}$, $\overline{x}$. However, this obtains* Empty *in many ways, as $[\underline{x}, \overline{x}]$ for any bounds satisfying $\underline{x} > \overline{x}$, and also as $[-\infty, -\infty]$ or $[+\infty, +\infty]$. The description (5) was preferred as it makes a one-to-one mapping between valid pairs $\underline{x}, \overline{x}$ of endpoints and the nonempty intervals they specify.*

]

A **box** or **interval vector** is an n-tuple $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$ whose components $\boldsymbol{x}_i$ are intervals, that is a member of $\overline{\mathbb{IR}}^n$. Usually $\boldsymbol{x}$ is identified with the cartesian product $\boldsymbol{x}_1 \times \ldots \times \boldsymbol{x}_n$ of its components,

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

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

10.4. Functions.

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

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

- A *point function* (§10.4.2) is a mathematical real function of real variables. Otherwise, *function* is usually used with its general mathematical meaning.
- A (point) *arithmetic operation* (§10.4.2) is a mathematical real function for which an implementation provides versions in the implementation's *library* (§10.4.2).
- A *version* of a point function f means a function derived from f; typically a bare or decorated interval extension (§10.4.3) of f.
- An *interval arithmetic operation* is an interval extension of a point arithmetic operation (§10.4.3).
- An *interval non-arithmetic operation* is an interval-to-interval library function that is not an interval arithmetic operation (§10.4.3).
- A *constructor* is a function that creates an interval from non-interval data (§10.6.9).

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

$$f(x_1, \ldots, x_n). \tag{6}$$

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

$$\mathrm{Rge}(f \mid \boldsymbol{s}) = \{\, f(x) \mid x \in \boldsymbol{s} \text{ and } x \in \mathrm{Dom}\, f \,\}. \tag{7}$$

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

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

A (point) **arithmetic operation** is a function for which an implementation provides versions in a collection of user-available operations called its **library**. This includes functions normally written in operator form (e.g., $+$, $\times$) and those normally written in function form (e.g., exp, arctan).

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

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

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

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

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

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

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

and the natural interval extension is defined by

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

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

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

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

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

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

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

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

10.6. 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 §10.6.2, 10.6.3, 10.6.4, 10.6.6, each such version shall be an interval extension (§10.4.3) of the corresponding point function—for a constant, that means any constant interval enclosing the point value. The required rounding behavior of these, and of the numeric functions of intervals in §10.6.10, is detailed in §12.9, 12.12.

The names of operations, as well as symbols used for operations (e.g., for the comparisons in §10.6.11), may not correspond to those that any particular language would use.

10.6.1. *Interval literals.*

An **interval literal** is a text string that denotes an interval. Level 1, which is mainly for human communication, merely assumes some agreed rules on the form and meaning of interval literals exist. 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]`.]

10.6.2. *Interval constants.*

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

10.6.3. *Forward-mode elementary functions.*

Table 10.1 on page 27 lists required arithmetic operations. The term *operation* includes functions normally written in function notation $f(x, y, \ldots)$, as well as those normally written in unary or binary operator notation, $\bullet x$ or $x \bullet y$.

[*Note. The list includes all general-computational operations in 754*§*5.4 except* **convertFromInt***, and some recommended functions in 754*§*9.2.*]

10.6.4. *Interval case expressions and case function.*

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

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

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

$$\mathtt{case}(\boldsymbol{c}, \boldsymbol{g}, \boldsymbol{h}) = \begin{cases} \text{if } \boldsymbol{c} \text{ is empty} & \text{then} & \emptyset \\ \text{elseif } \boldsymbol{c} \text{ is a subset of the half- line } x < 0 & \text{then} & \boldsymbol{g} \\ \text{elseif } \boldsymbol{c} \text{ is a subset of the half- line } x \geq 0 & \text{then} & \boldsymbol{h} \\ \text{else} & & \mathtt{convexHull}(\boldsymbol{g}, \boldsymbol{h}). \end{cases} \tag{8}$$

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

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

$$\boldsymbol{f}(\boldsymbol{x}) = \mathtt{case}\big(\boldsymbol{c}(\boldsymbol{x}), \boldsymbol{g}(\boldsymbol{x}), \boldsymbol{h}(\boldsymbol{x})\big) \tag{9}$$

is automatically an interval extension of f.

[*Notes.*

1. *Equation (8) does not define the natural interval extension, which has value Empty if* any *of its input arguments is empty. Its advantage is that for a function defined by a conditional expression, such as (9), it allows "short-circuiting". That is, one can suppress evaluation of* $\boldsymbol{h}(\boldsymbol{x})$ *if* $\overline{c} < 0$*, and of* $\boldsymbol{g}(\boldsymbol{x})$ *if* $\underline{c} \geq 0$*. This is not so for the natural extension.*
2. *This method is less awkward than using interval comparisons as a mechanism for handling such functions. However, the resulting interval function is usually not the tightest extension of the corresponding point function. E.g., the (point) absolute value* $|x|$ *may be defined by*

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

TABLE 10.1. Required forward elementary functions.
Normal mathematical notation is used to include or exclude an interval endpoint, e.g., $(-\pi, \pi]$ denotes $\{\, x \in \mathbb{R} \mid -\pi < x \leq \pi \,\}$.

Name	Definition	Point function domain	Point function range	Note
`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)	$(x \times y) + z$	$\mathbb{R}^3$	$\mathbb{R}$	
`case`(c,g,h)		See §10.6.4.		
`pown`(x,p)	x^p, $p \in \mathbb{Z}$	$\begin{cases}\mathbb{R} \text{ if } p \geq 0 \\ \mathbb{R}\setminus\{0\} \text{ if } p < 0\end{cases}$	$\begin{cases}\mathbb{R} \text{ if } p > 0 \text{ odd} \\ [0,\infty) \text{ if } p > 0 \text{ even} \\ \{1\} \text{ if } p = 0 \\ \mathbb{R}\setminus\{0\} \text{ if } p < 0 \text{ odd} \\ (0,\infty) \text{ if } p < 0 \text{ even}\end{cases}$	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
`sin`(x)		$\mathbb{R}$	$[-1,1]$	
`cos`(x)		$\mathbb{R}$	$[-1,1]$	
`tan`(x)		$\mathbb{R}\setminus\{(k+\frac{1}{2})\pi \mid k \in \mathbb{Z}\}$	$\mathbb{R}$	
`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)		$(-1,1)$	$\mathbb{R}$	
`sign`(x)		$\mathbb{R}$	$\{-1,0,1\}$	h
`ceil`(x)		$\mathbb{R}$	$\mathbb{Z}$	j
`floor`(x)		$\mathbb{R}$	$\mathbb{Z}$	j
`trunc`(x)		$\mathbb{R}$	$\mathbb{Z}$	j
`roundTiesToEven`(x)		$\mathbb{R}$	$\mathbb{Z}$	j
`roundTiesToAway`(x)		$\mathbb{R}$	$\mathbb{Z}$	j
`abs`(x)	$\lvert x \rvert$	$\mathbb{R}$	$[0,\infty)$	
`min`$(x_1,\ldots,x_k)$		$\mathbb{R}^k$ for $k = 2,3,\ldots$	$\mathbb{R}$	k
`max`$(x_1,\ldots,x_k)$		$\mathbb{R}^k$ for $k = 2,3,\ldots$	$\mathbb{R}$	k

Then it is easy to see that formula (9), applied to a nonempty $\boldsymbol{x} = [\underline{x}, \overline{x}]$*, gives the exact range* $\{\, |x| \mid x \in \boldsymbol{x} \,\}$ *when* $\overline{x} < 0$ *or* $0 \leq \underline{x}$*, but the poor enclosure* $(-\boldsymbol{x}) \cup \boldsymbol{x}$ *when* $\underline{x} < 0 \leq \overline{x}$*.*

3. `case`(c,g,h) *is equivalent to the C expression* $(c < 0\,?\,g : h)$*.*
4. *Compound conditions may be expressed using the* `max` *and* `min` *operations: e.g., a real function* $f(x,y)$ *that equals* $\sin(xy)$ *in the positive quadrant of the plane, and zero elsewhere, may be written*

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

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

]

10.6.5. *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 endpoint respectively. Then*

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

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 $\text{hull}(\boldsymbol{a})$.
[*Note.* `divToPair` *is not regarded as an arithmetic operation, since if it appears in an arithmetic expression, containment may be lost unless its two outputs are handled with care.*]

10.6.6. *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

$$\varphi\text{Rev}(\boldsymbol{c}, \boldsymbol{x}) \supseteq \{\, x \in \boldsymbol{x} \mid \varphi(x) \text{ is defined and in } \boldsymbol{c} \,\}, \tag{10}$$

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 $\bullet\text{Rev}_1$ and $\bullet\text{Rev}_2$ such that

$$\bullet\text{Rev}_1(\boldsymbol{b}, \boldsymbol{c}, \boldsymbol{x}) \supseteq \{\, x \in \boldsymbol{x} \mid b \in \boldsymbol{b} \text{ exists such that } x \bullet b \text{ is defined and in } \boldsymbol{c} \,\}, \tag{11}$$

$$\bullet\text{Rev}_2(\boldsymbol{a}, \boldsymbol{c}, \boldsymbol{x}) \supseteq \{\, x \in \boldsymbol{x} \mid a \in \boldsymbol{a} \text{ exists such that } a \bullet x \text{ is defined and in } \boldsymbol{c} \,\}. \tag{12}$$

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

In each of (10, 11, 12), 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 (10, 11, 12) 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 10.2. Note $\texttt{pownRev}(x, p)$ is regarded as a family of unary functions parametrized by p.
[*Example.*

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

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

TABLE 10.2. Required reverse elementary functions.

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

– *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.

]

10.6.7. *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 $\boldsymbol{s}_k = \sum_{i=1,\, i\neq k}^{n} \boldsymbol{a}_i$, for $k = 1, \ldots, n$.*
Evaluating all these sums independently costs $O(n^2)$ work. However, if one forms the sum $\boldsymbol{s}$ of all *the $\boldsymbol{a}_i$, one can obtain each $\boldsymbol{s}_k$ from $\boldsymbol{s}$ and $\boldsymbol{a}_k$ by cancellative subtraction. This method only costs $O(n)$ work.*

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

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 `cancelMinus`$(\boldsymbol{x},\boldsymbol{y})$ is the tightest interval $\boldsymbol{z}$ such that

$$\boldsymbol{y} + \boldsymbol{z} \supseteq \boldsymbol{x} \tag{13}$$

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

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

[*Note. Because of the cancellative nature of these operations, care is needed in finite precision to determine whether the result is defined or not. More details are given at Level 3 in §14.6.*]

10.6.8. *Set operations.*

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

10.6.9. *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 `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 `textToInterval`(s) takes a text string s. If s is a valid interval literal, see §10.6.1, 12.11, its value is the interval denoted by s. Otherwise it has no value.

10.6.10. *Numeric functions of intervals.*

The operations in Table 10.3 shall be provided, the argument being an interval and the result a number, which for some of the operations may be infinite.

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

TABLE 10.3. Required numeric functions of an interval $\boldsymbol{x} = [\underline{x}, \overline{x}]$.
Note `sup` can have value $-\infty$; each of `inf`, `wid`, `rad` and `mag` can have value $+\infty$.

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

10.6.11. *Boolean functions of intervals.*

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

There is a function `isEmpty`$(\boldsymbol{x})$, with value 1 if $\boldsymbol{x}$ is the empty set, 0 otherwise. There is a function `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 10.4, in which column three gives the set-theoretic definition, and column four gives an equivalent specification when both intervals are nonempty. Table 10.5 shows what the definitions imply when at least one interval is empty.

[*Notes.*

- *Column two of Table 10.4 gives suggested symbols for use in typeset algorithms.*
- *All these relations, except* $\boldsymbol{a} \between \boldsymbol{b}$*, 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.*
- *In fact all occurrences of* $<$ *in column 4 of Table 10.4 can be replaced by* $<'$*.*

]

TABLE 10.4. Comparisons for intervals $\boldsymbol{a}$ and $\boldsymbol{b}$. Notation $\forall_a$ means "for all a in $\boldsymbol{a}$", and so on. In column 4, $\boldsymbol{a}=[\underline{a},\overline{a}]$ and $\boldsymbol{b}=[\underline{b},\overline{b}]$, where $\underline{a},\underline{b}$ may be $-\infty$ and $\overline{a},\overline{b}$ may be $+\infty$; and $<'$ is the same as $<$ except that $-\infty <' -\infty$ and $+\infty <' +\infty$ are true.

Name	Symbol	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`$(\boldsymbol{a},\boldsymbol{b})$	$\boldsymbol{a}\prec\boldsymbol{b}$	$\forall_a \forall_b\, a\le b$	$\overline{a}\le\underline{b}$	$\boldsymbol{a}$ is to left of but may touch $\boldsymbol{b}$
`interior`$(\boldsymbol{a},\boldsymbol{b})$	$\boldsymbol{a}\circledcirc\boldsymbol{b}$	$\forall_a \exists_b\, a<b \wedge \forall_a \exists_b\, b<a$	$\underline{b}<'\underline{a} \wedge \overline{a}<'\overline{b}$	$\boldsymbol{a}$ 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`$(\boldsymbol{a},\boldsymbol{b})$	$\boldsymbol{a}\between\boldsymbol{b}$	$\forall_a \forall_b\, a\neq b$	$\overline{a}<\underline{b} \vee \overline{b}<\underline{a}$	$\boldsymbol{a}$ and $\boldsymbol{b}$ are disjoint

TABLE 10.5. 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
$\boldsymbol{a}\prec\boldsymbol{b}$	1	1	1
$\boldsymbol{a}\circledcirc\boldsymbol{b}$	1	0	1
$\boldsymbol{a}<\boldsymbol{b}$	0	0	1
$\boldsymbol{a}\prec\boldsymbol{b}$	1	1	1
$\boldsymbol{a}\between\boldsymbol{b}$	1	1	1

10.7. 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.

10.7.1. *Forward-mode elementary functions.*

The list of recommended functions is in Table 10.6. Each interval version shall be an interval extension of the point function.

Notes to Table 10.6

a. Regarded as a family of functions parameterized by the integer arguments q, or r and s.
b. $b = e, 2$ or 10, respectively.
c. Mathematically unnecessary, but included to let implementations give better numerical behavior for small values of the arguments.
d. In describing domains, notation such as $\{y = 0\}$ is short for $\{\,(x, y) \in \mathbb{R}^2 \mid y = 0\,\}$, and so on.
e. These functions avoid a loss of accuracy due to π being irrational, cf. Table 10.1, note e.
f. To avoid confusion with notation for open intervals, in this table coordinates in $\mathbb{R}^2$ are delimited by angle brackets $\langle\ \rangle$.

10.7.2. *Slope functions.*

The functions in Table 10.7 are the commonest ones needed to efficiently implement improved range enclosures via *first- and second-order slope* algorithms. They are analytic at $x = 0$ after filling in the removable singularity there, where each has the value 1.

10.7.3. *Extended interval comparisons.*

The **interval overlapping function** `overlap`$(\boldsymbol{a},\boldsymbol{b})$, also written $\boldsymbol{a}\operatorname{\textcircled{o}}\boldsymbol{b}$, 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.

TABLE 10.6. Recommended elementary functions.
Normal mathematical notation is used to include or exclude an interval endpoint, e.g., $(-1, 1]$ denotes $\{\, x \in \mathbb{R} \mid -1 < x \leq 1 \,\}$.

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

TABLE 10.7. Recommended slope functions.

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

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

To describe the states for nonempty intervals of positive width, it is useful to think of $\boldsymbol{b} = [\underline{b}, \overline{b}]$ (with $\underline{b} < \overline{b}$) as fixed, while $\boldsymbol{a} = [\underline{a}, \overline{a}]$ (with $\underline{a} < \overline{a}$) starts far to its left and moves to the right. Its endpoints move continuously with strictly positive velocity. Then, depending on the relative sizes

of $\boldsymbol{a}$ and $\boldsymbol{b}$, the value of $\boldsymbol{a} \circledcirc \boldsymbol{b}$ follows a path from left to right through the graph below, whose nodes represent Allen's 13 states.

$$\begin{array}{ccccccccccc} & & & & \text{starts} & \to & \text{containedBy} & \to & \text{finishes} & & \\ & & & & \uparrow & & & & \downarrow & & \\ \to & \text{before} & \to & \text{meets} & \to & \text{overlaps} & \to & \text{equals} & \to & \text{overlappedBy} & \to & \text{metBy} & \to & \text{after} & \to \\ & & & & \downarrow & & & & \uparrow & & \\ & & & & \text{finishedBy} & \to & \text{contains} & \to & \text{startedBy} & & \end{array}$$

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

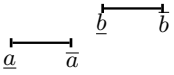
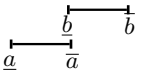
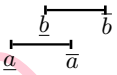
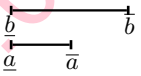
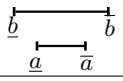
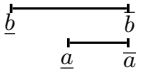
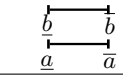
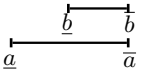
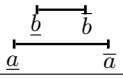
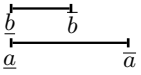
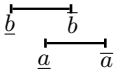
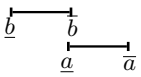
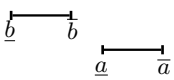
The set and endpoint specifications remove some ambiguities of the diagram view when one interval shrinks to a single point that coincides with an endpoint of the other. Such a case is allocated to `equal` when all four endpoints coincide; else to `starts`, `finishes`, `finishedBy` or `startedBy` as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

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

For instance, suppose we make the states s in Table 10.8'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.*]

TABLE 10.8. The 16 states of interval overlapping situations for intervals $\boldsymbol{a}, \boldsymbol{b}$. Notation $\forall_a$ means "for all a in $\boldsymbol{a}$", and so on. Phrases within a cell are joined by "and", e.g. `starts` is specified by $(\underline{a} = \underline{b} \wedge \overline{a} < \overline{b})$.

State $\boldsymbol{a} \circledcirc \boldsymbol{b}$ is	Set specification	Endpoint specification	Diagram
States with either interval empty			
`bothEmpty`	$\boldsymbol{a} = \emptyset \wedge \boldsymbol{b} = \emptyset$		
`firstEmpty`	$\boldsymbol{a} = \emptyset \wedge \boldsymbol{b} \neq \emptyset$		
`secondEmpty`	$\boldsymbol{a} \neq \emptyset \wedge \boldsymbol{b} = \emptyset$		
States with both intervals nonempty			
`before`	$\forall_a \forall_b\ a < b$	$\overline{a} < \underline{b}$	
`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`	$\forall_b \forall_a\ b < a$	$\overline{b} < \underline{a}$	

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 how decorations are bound to library interval arithmetic operations to give correct propagation through expressions. §11.7, 11.8 are 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. §11.9 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; §11.10 gives explanatory notes on the `com` decoration. §11.11 defines a restricted decorated arithmetic that suffices for some important applications and is easier to implement efficiently.

In Annex D, §D.2 gives examples of the meaning and use of decorations; and §D.4 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 $n \geq 0$ and $\boldsymbol{x} \in \overline{\mathbb{IR}}^n$ is an n-dimensional box, regarded as a subset of $\mathbb{R}^n$. The notation $(f, \boldsymbol{x})$ unless said otherwise denotes such a pair, for arbitrary n, f and $\boldsymbol{x}$. Equivalently, d is identified with the set of pairs for which the property holds:

$$d = \{\, (f, \boldsymbol{x}) \mid p_d(f, \boldsymbol{x}) \text{ is true} \,\}. \tag{14}$$

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.

(15)

These are listed according to the propagation order (24), which may also be thought of as a quality-order of $(f, \boldsymbol{x})$ pairs—decorations above `trv` are "good" and those below are "bad".

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

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

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

Treating the decorations as sets as in (14), `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. (16)

Namely the definitions (15) give:

$$\texttt{com} \subset \texttt{dac} \subset \texttt{def} \subset \texttt{trv} \supset \texttt{ill}, \qquad \text{note the change from } \subset \text{ to } \supset; \tag{17}$$

$$\texttt{com}, \texttt{dac} \text{ and } \texttt{def} \text{ are disjoint from } \texttt{ill}. \tag{18}$$

Property (16) implies that for any $(f, \boldsymbol{x})$ there is a unique tightest (in the containment order (17)), 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 $\mathrm{dec}(f, \boldsymbol{x})$. That is:

$$\mathrm{dec}(f, \boldsymbol{x}) = d \iff p_d(f, \boldsymbol{x}) \text{ holds, but } p_e(f, \boldsymbol{x}) \text{ fails for all } e \subset d. \tag{19}$$

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

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 $\mathrm{Dom}(f) = \emptyset$ of $(f, \boldsymbol{x})$ pairs, see Clause E.3.

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

Other information may be stored in an NaI in an implementation-defined way (like the payload of a 754 floating-point NaN), and functions may be provided for a user to set and read this for diagnostic purposes. An implementation may also provide means for an exception to be 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 $\boldsymbol{y} \supseteq \mathrm{Rge}(f \,|\, \boldsymbol{x})$ 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 $dy =$ `dac` or `def` then by definition $\boldsymbol{x}$ is nonempty, and f is everywhere defined on it, so that $\mathrm{Rge}(f \,|\, \boldsymbol{x})$ is nonempty, implying $\boldsymbol{y}$ is nonempty. Hence the decorated intervals $\emptyset_{\texttt{dac}}$ and $\emptyset_{\texttt{def}}$, and $\emptyset_{\texttt{com}}$ if `com` is provided, are contradictory: implementations shall not produce them.

No other combinations are essentially forbidden.

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.

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, $\mathrm{Id}(x) = x$. 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 $p_d(\mathrm{Id}, \boldsymbol{x})$ true, that is

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

The function `newDec`() initializes a bare interval in this way:

$$\texttt{newDec}(\boldsymbol{x}) = \boldsymbol{x}_d \quad \text{where} \quad d = \mathrm{dec}(\mathrm{Id}, \boldsymbol{x}). \tag{20}$$

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

Disassembling and assembling. 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. NaI is deemed to equal $\emptyset_{\texttt{ill}}$, so that $\texttt{intervalPart}(\text{NaI}) = \emptyset$ and $\texttt{decorationPart}(\text{NaI}) = \texttt{ill}$.

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

[*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 may need it, e.g., to decorate the output of functions whose definition involves the* `intersection` *and* `convexHull` *operations.*]

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 (24).

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

$$\boldsymbol{w} \supseteq \text{Rge}(\varphi \,|\, \boldsymbol{v}), \qquad \text{(enclosure)}. \tag{21}$$

The decorated interval extension of φ determines a dv_0 such that

$$p_{dv_0}(\varphi, \boldsymbol{v}) \text{ holds}, \qquad \text{(a "local decoration")}. \tag{22}$$

It then evaluates the output decoration dw by

$$dw = \min\{dv_0, dv_1, \ldots, dv_k\}, \qquad \text{(the "min-rule")}, \tag{23}$$

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

$$\texttt{com} > \texttt{dac} > \texttt{def} > \texttt{trv} > \texttt{ill}. \tag{24}$$

[*Notes.*

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

   $$\boldsymbol{y} \supseteq \text{Rge}(f \,|\, \boldsymbol{x}) \tag{25}$$

   but also

   $$p_{dy}(f, \boldsymbol{x}) \text{ holds}. \tag{26}$$

   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 D.*

   Briefly, (20) 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 (23) 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 (21) is compatible with many bare interval extensions, typically coming from different interval types at Level 2, so there may be several* dv_0 *satisfying the local decoration requirement (22). The ideal choice is the strongest decoration* d *such that* $p_d(\varphi, \boldsymbol{v})$ *holds, that is to take*

   $$dv_0 = \text{dec}(\varphi, \boldsymbol{v}). \tag{27}$$

This is easily computable in finite precision for the arithmetic operations in §10.6, 10.7—see the tables in Annex D, §D.1. However, functions may be added to the library in future for which (27) is impractical to compute for some arguments $\boldsymbol{v}$. Hence the weaker requirement (22) 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 §10.6.6;
– the cancellative operations `cancelPlus`$(\boldsymbol{x},\boldsymbol{y})$ and `cancelMinus`$(\boldsymbol{x},\boldsymbol{y})$ of §10.6.7;
– The set-oriented operations `intersection`$(\boldsymbol{x},\boldsymbol{y})$ and `convexHull`$(\boldsymbol{x},\boldsymbol{y})$ of §10.6.8.

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 §10.6.10;
– the boolean-valued functions of §10.6.11;
– the overlap function of §10.7.3.

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

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

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

11.9. 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 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 (23). In this case it is straightforward to compute the strongest local decoration $d = \mathsf{dec}(f, \boldsymbol{x})$*, as follows.*

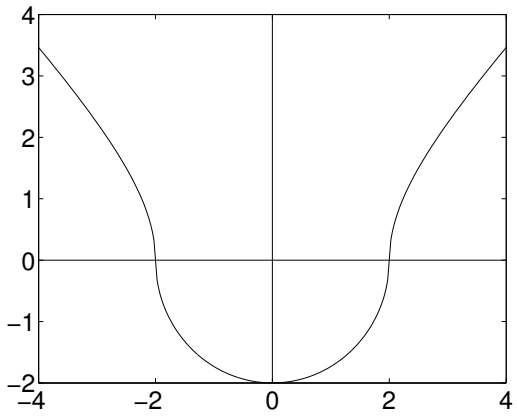
$$d = \begin{cases} \texttt{com} & \text{if } 0 \notin \boldsymbol{x} \text{ and } \boldsymbol{x} \text{ is nonempty and bounded,} \\ \texttt{dac} & \text{if } 0 \notin \boldsymbol{x} \text{ and } \boldsymbol{x} \text{ is unbounded,} \\ \texttt{trv} & \text{if } \boldsymbol{x} = \emptyset \text{ or }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

(ii)

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

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

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



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

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

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

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.*

]

11.10. Notes on the com decoration.

[*Notes.*

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

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

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

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

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

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

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

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

]

11.11. Compressed arithmetic with a threshold (optional).

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

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

The context is that of evaluating an arithmetic expression, where the use made of a decorated interval evaluation $\boldsymbol{y}_{dy} = f(\boldsymbol{x}_{dx})$ depends on a check of the result decoration dy against an application-dependent **exception threshold** τ, where $\tau \geq \texttt{trv}$ in the propagation order (24):

- $dy \geq \tau$ represents normal computation. The decoration is not used, but one exploits the range enclosure given by the interval part and the knowledge that dy remained $\geq \tau$.
- $dy < \tau$ declares an exception to have occurred. The interval part is not used, but one exploits the information given by the decoration.

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

– Only decorations $< \tau$ occur; in particular com is never used.
– The empty interval $\emptyset$ is replaced by—equivalently, is regarded by the implementation as being—a new decoration emp added to the table in (15), 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;

(28)

emp lies between trv and ill in the containment order (17) and the propagation order (24):

$$\begin{aligned} &\texttt{com} \subset \texttt{dac} \subset \texttt{def} \subset \texttt{trv} \supset \texttt{emp} \supset \texttt{ill}, \\ &\texttt{com} > \texttt{dac} > \texttt{def} > \texttt{trv} > \texttt{emp} > \texttt{ill}. \end{aligned} \tag{29}$$

Since $\tau \geq \texttt{trv}$ it is always true that $\texttt{emp} < \tau$, which means that as soon as an empty result is produced while evaluating an expression, the $dy < \tau$ case has occurred.

[*Note. The reason for treating $\emptyset$ as a decoration $< \tau$ is that obtaining an empty result (e.g., by doing something like $\sqrt{[-2, -1]}$ while evaluating a function) is one of the "exceptions" that compressed interval computation should detect.*]

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

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

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

11.11.3. *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 ≥ τ, return the 𝕋τ-interval with value x
else return the 𝕋τ-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 x is an interval, return (x, τ).
if x is a decoration d
  if d is ill or emp, return (Empty, d)
  else return (Entire, d).
```

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

– Operations purely on bare intervals are performed as if each $\boldsymbol{x}$ is the decorated interval $\boldsymbol{x}_\tau$, resulting in a decorated interval $\boldsymbol{y}_{dy}$ that is then converted back into a compressed interval. If $dy < \tau$, the result is the bare decoration dy, otherwise the bare interval $\boldsymbol{y}$.
– For operations with at least one bare decoration input, the result is always a bare decoration. A bare interval input is treated as in the previous item. A decoration d in {`emp`, `ill`} is treated as $\emptyset_d$. A decoration d in {`trv`, `def`, `dac`} is treated (conceptually) as $\boldsymbol{x}_d$ with an arbitrary nonempty interval $\boldsymbol{x}$. The decoration `com` cannot occur. Performing the resulting decorated interval operation on all such possible inputs leads to a set of all possible results $\boldsymbol{y}_{dy}$. The tightest decoration (in the containment order (29)) enclosing all resulting dy is returned.

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

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

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

(30)

[*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. In Annex D, sample tables and worked examples are in Clause D.3 and a proof of correctness of the compressed arithmetic system is in Clause D.5.

If compressed arithmetic is implemented, it shall provide versions of all the required operations of §10.6, and it should provide the recommended operations of §10.7.

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.

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.

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 §11.11 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 support at least one bare interval type. If 754-conforming, it shall support 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 supported is language- or implementation-defined.

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

A Level 2 operation is a finite-precision approximation to the corresponding Level 1 operation. Whereas a Level 1 operation, for some inputs, may have no value (be undefined in the mathematical sense), each Level 2 operation shall return a value for arbitrary inputs.

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 $z = x + y$ where each of x, y, 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 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 §10.6, 10.7.
(b) A set operation, such as intersection and convex hull of $\mathbb{T}$-intervals, returning a $\mathbb{T}$-interval.
(c) A function such as the midpoint, whose input is a $\mathbb{T}$-interval and output is numeric.
(d) A constructor, whose input is numeric or text and output is a $\mathbb{T}$-datum.
(e) The $\mathbb{T}$-interval hull, regarded as a conversion operation, see §12.12.11.

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

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 $\boldsymbol{x}, \boldsymbol{y}$ may 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`$(\boldsymbol{x}, \boldsymbol{y})$.

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

It may 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 may 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" 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 64 bit binary is a different datum from the same number represented in 32 bit binary (`double` or `float` respectively, in C).

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 some 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 §10.6.11, 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 **equal as 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, it returns `true` and they are called equal. If their types are not comparable, `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. As datums they are unequal; but $x = y$ (equivalently* `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 $[1, 3.75]$. Then they are unequal as datums; but $\boldsymbol{x} = \boldsymbol{y}$ (equivalently* `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. Number formats describe data, usually floating-point or integer, that may be input to, or output from, operations of this standard. In view of §12.3, a **number format**, or just format, is formally the set of all pairs (x, f) where x belongs to a set $\mathbb{F}$, and f is a name for the format.

$\mathbb{F}$ comprises a finite subset of the extended reals $\overline{\mathbb{R}}$, together with a value NaN. $\mathbb{F}$ shall contain $-\infty$, $+\infty$, zero, and at least one nonzero finite number. It shall be symmetric: if a real x is in $\mathbb{F}$, so is $-x$. Signed zeros, that is –0 and +0 either replacing or distinct from 0, may exist, depending on the format.

Following the convention of omitting names, the format is normally identified with the set $\mathbb{F}$, and one may say a number format is a set of datums comprising NaN together with a finite subset of $\overline{\mathbb{R}}$ (and possible ± 0) subject to the above rules. A member of $\mathbb{F}$ is called an **$\mathbb{F}$-datum**. It is **numeric**, or an **$\mathbb{F}$-number**, if it is not NaN.

A floating-point format in the 754 sense, such as `binary64`, is identified with the number format for which $\mathbb{F}$ is the set of datums representable in that 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.

[*Note. At Level 2 each format has only one zero datum and one* NaN *datum, but these may correspond to several values at Level 3, e.g. IEEE 754 has zeros* $-0, +0$*, while* NaN*s are distinguished by sign bit and payload, and by being quiet or signaling. The fact that a floating-point operation at Level 2 can behave differently on nominally equal datums is beyond the remit of this standard. E.g.,* -0 *and* $+0$ *compare equal but* $1/-0$ *and* $1/+0$ *do not.*]

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 an operation producing a numeric result, that has a Level 2 version with output of some format $\mathbb{F}$, the Level 2 result is obtained by mapping (rounding) the Level 1 result to an $\mathbb{F}$-number according to rules specific to that operation.

12.5. Bare and decorated interval types.

12.5.1. *Definition.* In view of §12.3, a **bare interval type** is formally the set of all pairs $(\boldsymbol{x}, t)$ where $\boldsymbol{x}$ belongs to a finite subset $\mathbb{T}$ of the mathematical intervals $\overline{\mathbb{IR}}$ that contains Empty and Entire, and t is a name for the type. The **decorated interval type derived** from $\mathbb{T}$ is formally the set of triples $(\boldsymbol{x}, t, d)$ where $(\boldsymbol{x}, t)$ 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 finite set $\mathbb{T}$ of intervals that contains Empty and Entire. 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 given 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 endpoints are in $\mathbb{F}$, together with Empty. When $\mathbb{F}$ is a 754 format, the **radix** of $\mathbb{T}$ means the radix of $\mathbb{F}$. Note that Entire is

[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.

in $\mathbb{T}$ because $\pm\infty \in \mathbb{F}$ by the definition of a number format, so $\mathbb{T}$ satisfies the requirements for a bare interval type given in §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 is in some number format $\mathbb{F}$, and r is in a possibly different number format $\mathbb{F}'$, with m, r finite and $r \geq 0$. From the definition in §12.5.1 such a type shall contain Empty and Entire (so at Level 3 it shall have representations of these). Whether such a type also contains semi-bounded intervals is language- or implementation-defined.

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 Table 12.1.

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 may 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. This eliminates the problem of double rounding in mixed-format work.

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 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 Table 12.1.

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

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

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

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* $\mathsf{hull}_{\mathbb{T}}$ *as the operation* `convertType` *for conversion between any two supported types, see §12.12.11.*]

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 $\boldsymbol{y}$ may not be unique. If it is unique for every subset $\boldsymbol{s}$ of $\mathbb{R}$, then the type $\mathbb{T}$ is called **explicit**, otherwise it is **implicit**.

Two types with different hull operations are different, even if they have the same set of intervals. [*Examples. Every inf-sup type is explicit. A mid-rad type is typically implicit.*

As an example of the need for a specified hull algorithm, let $\mathbb{T}$ *be the mid-rad type (§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 $\mathrm{hull}_{\mathbb{T}}(\boldsymbol{s}) = (\boldsymbol{y}_1, \ldots, \boldsymbol{y}_n)$ where

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

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

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

$$\boldsymbol{f}(\boldsymbol{x}) \supseteq \mathrm{Rge}(f \mid \boldsymbol{x}). \tag{31}$$

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

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

for any subset $\boldsymbol{s}$ of $\mathbb{R}^n$. Then the interval $\boldsymbol{f}^*(\boldsymbol{s})$ contains $\mathrm{Rge}(f \mid \boldsymbol{s})$ 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. For each required forward or reverse operation on a 754-conforming interval type, the accuracy shall be as listed in Table 12.1. For such an operation on any other inf-sup type, the accuracy listed in Table 12.1 is recommended. For all other operations, the accuracy is language- or implementation-defined. For all operations, on all types, the accuracy achieved by the implementation shall be documented.

In this subclause, *operation* denotes any Level 2 version, provided by the implementation, of a Level 1 operation with interval output and at least one interval input. Bare interval operations are described; the accuracy of a decorated operation is defined to be that of its bare 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 §10.6.3, 10.6.6 (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 and, for a forward or reverse arithmetic operation, let f be the underlying point function[3].

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

$$\boldsymbol{f}(\boldsymbol{x}) \supseteq \boldsymbol{f}_{\text{exact}}(\boldsymbol{x}). \tag{32}$$

For a forward arithmetic operation, it is equivalent to (31).

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

$$\boldsymbol{f}_{\text{tightest}}(\boldsymbol{x}) = \text{hull}_{\mathbb{T}}(\boldsymbol{f}_{\text{exact}}(\boldsymbol{x})). \tag{33}$$

For a forward arithmetic operation, it is equivalent to

$$\boldsymbol{f}(\boldsymbol{x}) = \text{hull}_{\mathbb{T}}(\text{Rge}(f \,|\, \boldsymbol{x})). \tag{34}$$

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*, (32), and is at most slightly wider than the result of applying the *tightest* version to a slightly wider input box:

$$\boldsymbol{f}(\boldsymbol{x}) \subseteq \texttt{nextOut}(\boldsymbol{f}_{\text{tightest}}(\texttt{nextOut}(\text{hull}_{\mathbb{T}}(\boldsymbol{x})))), \tag{35}$$

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

$$\texttt{nextOut}(\boldsymbol{x}) = [\texttt{nextDown}(\underline{x}), \texttt{nextUp}(\overline{x}). \tag{36}$$

For a $\mathbb{T}$-box, `nextOut` acts componentwise.

[*Notes.*

- *In (35), the inner* `nextOut()` *aims to handle the problem of a function like* $\sin x$ *evaluated at a very large argument, where a small relative change in the input can produce a large relative change in the result. The outer* `nextOut()` *relaxes the requirement for correct (rather than, say, faithful) rounding, which may be hard to achieve for some special functions at some arguments.*
- *The 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* $\text{hull}_{\mathbb{T}}$ *in (35) 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 endpoint.*

  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*

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

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

]

12.10.2. *Table of accuracies.* For the operations in Table 12.1, for arbitrary interval inputs—including mixed-type inputs where relevant—and for each relevant operation, at least the listed tightness shall be achieved by any 754-conforming type, and should be achieved by any other inf-sup type.

For the `convertType` operation, converting between 754-conforming types, *tightest* is required; in other cases *tightest* is recommended, and *accurate* is required.

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

(a) Forward

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

(b) Reverse

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

(c) Two-output division

Name	Accuracy
`divToPair`(x, y)	tightest

TABLE 12.1. Accuracy levels for arithmetic operations.

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 may 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` §12.12.8 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 §10.6.1. However the value of $\boldsymbol{s}$ is a bare or decorated Level 1 interval $\boldsymbol{x}$. Within $\boldsymbol{s}$, conversion of a number literal to its value 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 endpoints 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 supported.

(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 optional 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 number literals with $l \le u$, $l < +\infty$ and $u > -\infty$, see §10.2 and the note on difficulties of implementation §12.12.8. Its bare value is the mathematical interval $[l, u]$. A string `[` x `]` is equivalent to `[` x `,` x `]`.

(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 $\texttt{ulp}(m)$, 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.*

Decorated intervals literals may denote either bare or 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 $\texttt{newDec}(\boldsymbol{x})$ §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 12.2 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.

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 12.3, 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` §12.12.8, 13.2 of any implementation shall accept any portable interval. 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.

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

TABLE 12.2. Portable interval literal examples.

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.

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

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 §12.11 to §12.12.10, for each supported type $\mathbb{T}$. That is, those of its inputs and outputs that are intervals, are of type $\mathbb{T}$.

Operations in this subclause are described as functions with zero or more input arguments and one return value. It is language-defined whether they are implemented in this way: for instance two-output division, described in §12.12.4 as a function returning an ordered pair of intervals, might be implemented as a procedure `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, sign, ceil, floor, round, trunc, abs, min, max, fma`.

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

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 §10.6.2, returning a $\mathbb{T}$-interval with value Empty and Entire repectively. There shall also be a decorated version of each, returning `newDec`(Empty) = $\text{Empty}_{\texttt{trv}}$ and `newDec`(Entire) = $\text{Entire}_{\texttt{dac}}$ 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 §10.6.3. 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* `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 (22) 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 (22).

12.12.3. *Interval case expressions and case function.* An implementation shall provide the interval `case`$(\boldsymbol{c}, \boldsymbol{g}, \boldsymbol{h})$ function, see §10.6.4, for each supported type $\mathbb{T}$. The input $\boldsymbol{c}$ is of an arbitrary supported interval type. The inputs $\boldsymbol{g}, \boldsymbol{h}$, and the result, are of type $\mathbb{T}$. The implementation shall be as if the $\mathbb{T}$-version of `convexHull` is used in (8) of §10.6.4.

⚠ The decorated version is TBW. Arnold Neumaier had a special recipe, which I need to look up.

12.12.4. *Two-output division.*

There shall be a $\mathbb{T}$-version of the two-output division `divToPair`$(\boldsymbol{x}, \boldsymbol{y})$ of §10.6.5, for each supported bare interval type $\mathbb{T}$, namely

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

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`.

12.12.5. *Reverse-mode elementary functions.* An implementation shall provide a $\mathbb{T}$-version of each reverse arithmetic operation in §10.6.6, 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.

12.12.6. *Cancellative addition and subtraction.* An implementation shall provide a $\mathbb{T}$-version of each of the operations `cancelMinus` and `cancelPlus` in §10.6.7, 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 $\texttt{cancelMinus}(\boldsymbol{x}, \boldsymbol{y})$, 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 $\texttt{width}(\boldsymbol{x}) < \texttt{width}(\boldsymbol{y})$.

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

If $\mathbb{T}$ is a 754-conforming type, the result should be the $\mathbb{T}$-hull of the Level 1 result when this is defined.

[*Notes. Obtaining an accurate result may require computing in extra precision in extreme cases, namely where* $\boldsymbol{x}$ *and* $\boldsymbol{y}$ *are large intervals of nearly equal width. For an example let* $\mathbb{T}$ *be the inf-sup type derived from a format* $\mathbb{F}$*, and let* m *be the smallest positive and* M *the largest finite* $\mathbb{F}$*-number. Consider the cases where* $\boldsymbol{x}$ *and* $\boldsymbol{y}$ *are* $[-m, M]$ *and* $[m, M]$*, or vice versa.*

For any $\boldsymbol{x}, \boldsymbol{y}$*, the Level 1 result of these operations, if defined, is always bounded but might overflow at Level 2: that is, there might be no bounded* $\mathbb{T}$*-interval containing it. An example is* $\texttt{cancelMinus}([M, M], [-M, -M])$*. The mathematical value is* $[2M, 2M]$ *whose* $\mathbb{T}$*-hull is* $[M, +\infty]$*.*

An implementation should not, and for an inf-sup type it need not, return Entire *in the case of overflow. By avoiding doing so, it makes the result* Entire *diagnostic: it occurs if, and only if, the function has no value at Level 1.*]

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

12.12.7. *Set operations.* An implementation shall provide a $\mathbb{T}$-version of each of the operations `intersection` and `convexHull` in §10.6.8, 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.

[*Note. If* $\mathbb{T}$ *is an inf-sup type, the operation can always return the exact result. However, this need not be the case with a mixed-type version.*]

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

12.12.8. *Constructors.* For each supported bare or decorated interval type $\mathbb{T}$, there shall be a $\mathbb{T}$-version of each constructor in §10.6.9. It returns a $\mathbb{T}$-datum.

Difficulties in implementation. Both `numsToInterval`, and `textToInterval` when its input is a literal of inf-sup form, 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. In the former case, l and u are values of supported number formats within a program; in the latter, they are number literals.

Evaluating b as 0 when the true value is 1 (a "false negative") leads to falsely returning Empty as an enclosure of the true nonempty interval—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 supported 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 §10.6.9, 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) may occur shall be documented.

In case (a)—that is, neither l nor u is NaN, and the exact extended-real values of l and u are known to satisfy $l \leq u$, $l < +\infty$, $u > -\infty$—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 $\mathbb{T}$, the second case 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) may 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`.
- when $\boldsymbol{s}$ is of inf-sup form `[l,u]`, then dy shall equal `com`, except if $\boldsymbol{y}$ is unbounded because of overflow. Then dy shall equal `dac`.
- when $\boldsymbol{s}$ is of inf-sup form `[l,u]_dx`, then dy shall equal dx, except if $\boldsymbol{y}$ is unbounded because of overflow. Then dy shall equal `dac`.

Exception behavior. Exceptions denoted `IntvlConstructorFails` and `IntvlConstructorUnsure` shall be provided. `IntvlConstructorFails` 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.*] `IntvlConstructorUnsure` 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).

12.12.9. *Numeric functions of intervals.* An implementation shall provide a $\mathbb{T}$-version of each numeric function in Table 10.3 of §10.6.10, for each supported bare interval type $\mathbb{T}$. It shall return a result in a 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 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	The result 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.

– `rad`($\boldsymbol{x}$) returns NaN if $\boldsymbol{x}$ is empty, and otherwise the smallest $\mathbb{F}$-number r such that $\boldsymbol{x}$ is contained in the exact interval $[m - r, m + r]$, where m is the value returned by `mid`($\boldsymbol{x}$).
[*Note.* `rad`($\boldsymbol{x}$) *may be* $+\infty$ *even though* $\boldsymbol{x}$ *is bounded, if* $\mathbb{F}$ *has insufficient range. However, if* $\mathbb{F}$ *is a 754 format and* $\mathbb{T}$ *is the derived inf-sup type,* `rad`($\boldsymbol{x}$) *is finite for all bounded nonempty intervals.*]
– `wid`($\boldsymbol{x}$) returns 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 one of 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.

12.12.10. *Boolean functions of intervals.*

An implementation shall provide a $\mathbb{T}$-version of the function `isEmpty`($\boldsymbol{x}$) and the function `isEntire`($\boldsymbol{x}$) in §10.6.11, for each supported bare interval type $\mathbb{T}$.

An implementation shall provide a $\mathbb{T}$-version of each of the comparison relations in Table 10.4 of §10.6.11, 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 one of the corresponding decorated interval type. Following §11.7, if any input is NaI, the result is `false`. (In particular `equal`(NaI, NaI) returns `false`.) Otherwise the result is obtained by discarding the decoration and applying the corresponding bare interval operation.

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`.

12.12.11. *Interval type conversion.* An implementation shall provide the operation `convertType` to convert between any two supported bare interval types, and between any two supported decorated interval types. Conversion of a bare interval $\boldsymbol{x}$ of any type to an interval of type $\mathbb{T}$ is done by applying the $\mathbb{T}$-hull operation, see §12.8.1. That is, $\boldsymbol{y} = \mathbb{T}$-`convertType`$(\boldsymbol{x})$ is defined by

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

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

Conversion of a decorated interval is done by converting the interval part and leaving the decoration unchanged, *except* that if the decoration is `com` and the conversion overflows (produces an unbounded interval) the decoration is changed to `dac`. That is, if $\mathbb{DT}$ is the decorated type of $\mathbb{T}$ then $\boldsymbol{y}_{dy} = \mathbb{DT}$-`convertType`$(\boldsymbol{x}_{dx})$ is defined by

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

12.12.12. *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`.

12.12.13. *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.

(Note that these rules allow for short-circuit evaluation in certain cases.)

All other behavior, such as overflow, underflow, setting of IEEE 754 flags, raising of 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 §14.7.

12.13. Recommended operations.

12.13.1. *Forward-mode elementary functions.*

The functions listed in §10.7.1 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 §10.7.2 shall be handled in the same way as those in §12.13.1.

12.13.3. *Extended interval comparisons.*

How the operations in §10.7.3 are handled at Level 2 is implementation-defined.

12.14. Compressed arithmetic at Level 2.

How the operations in §11.11 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 may 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 §12.12.8. 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 §12.12.8. 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

`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]`*. Note that such a string is not a portable literal* §*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 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 equality as datums:* $\boldsymbol{x}$ *and* $\boldsymbol{y}$ *have the same Level 1 value and the same type. They may differ at Level 3, e.g., a zero endpoint 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. Note that 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 aim to display the values of the parameters that define the underlying mathematical model, in a human-readable way.

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`, and (see §12.12.8), shall return Empty and signal a language- or implementation-defined exception when its input is invalid.

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* $d_0 \ne 0$*.*]

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 $d_0.d_1d_2 \dots d_{p-1}$, 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 $d_0.d_1d_2 \dots d_{p-1}$, abbreviated in the same way, and then converted to a hexadecimal string $D_0.D_1 \dots$ (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 can be written* `0.000001e-95`.]

Above, alphabetic characters have been written in lowercase, but 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

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

Then

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

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* `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 strings, and the set $\mathbb{D}$ of decorations 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* `binary64` *may denote either the set of representations (in the sense of IEEE 754 §3.2) of binary64 numbers, or the set of numbers thus represented. Then for instance* -0 *and* $+0$ *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. (37)

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. Two possible representations are:*

- **inf-sup** *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 the object* $(\underline{x}, \overline{x})$*, and (for instance—many other ways are possible)* Empty *is represented by* (NaN, NaN)*. Its valid objects are* (NaN, NaN)*, together with all* (l, u) *such that* l, u *are not* NaN *and* $l \leq u$*,* $l < +\infty$*,* $u > -\infty$*.*
- **neginf-sup** *form. This is as the previous, except that* $\boldsymbol{x} = [\underline{x}, \overline{x}]$ *is represented by* $(-\underline{x}, \overline{x})$*. Its valid objects are* (NaN, NaN)*, together with all* (l, u) *such that* l, u *are not* NaN *and* $0 \leq l + u$*,* $l > -\infty$*,* $u > -\infty$*.*

If, in these descriptions l*,* u *and* NaN *are viewed as Level 2 datums, then each interval has only one representative: for a nonempty* $\mathbb{T}$*-interval there are unique* l *and* u*, and there is a unique* NaN *to use in the fields of* Empty*. However* l*,* u *and* NaN *themselves have representations, and from this viewpoint some intervals have more than one representative:* $[0, 1]$ *can be either* $(-0, 1)$ *or* $(+0, 1)$*, while there are many* NaN*s, 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 an exception to be signaled when this occurs.

To promote reproducibility, an implementation should provide a computational mode where, at least for library operations with numeric output, the representative of the output is independent of the representatives of the inputs. That is, in the notation above, y' is not changed by changing the x'_i. [*Example. Let* $\mathbb{F}$ *be a 754 format and* $\mathbb{T}$ *the derived inf-sup type. Suppose a nonempty* $\mathbb{T}$*-interval* $[l, u]$ *is represented at Level 3 as the pair of* $\mathbb{F}$*-numbers* (l, u)*. Let* f *be the expression*

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

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

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

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

14.4. 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.

14.5. Interchange formats. ⚠ We need a motion on this subclause, which was JDP's invention.

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

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

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

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

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

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

14.6. Operation tables for basic interval operations.

The tables in this subclause are an explicit realization of the general definition of interval operations given in §10.4.3. 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 $\mathbb{F}$:

$\bigtriangledown$, $\bigtriangleup$: the roundings downwards and upwards to the next element of $\mathbb{F}$,
$\overline{\bigtriangledown}$, etc. : the operations for elements of $\mathbb{F}$ with rounding downwards,
$\overline{\bigtriangleup}$, etc. : the operations for elements of $\mathbb{F}$ with rounding upwards.
$\diamondsuit\, \boldsymbol{s}$: 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:

$$\boldsymbol{a} \circ \boldsymbol{b} := \{\, a \circ b \mid a \in \boldsymbol{a} \ \wedge\ b \in \boldsymbol{b} \ \wedge\ a \circ b \text{ is defined} \,\}, \tag{38}$$

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:

$$a \diamondsuit b := \diamondsuit\,(a \circ b), \tag{39}$$

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

$$a \diamondsuit b = \left[\min_{i,j=1,2} (a_i \mathbin{\overline{\bigtriangledown}} b_j),\ \max_{i,j=1,2} (a_i \mathbin{\overline{\bigtriangleup}} b_j) \right],$$

or with the monotone roundings $\bigtriangledown$ and $\bigtriangleup$,

$$a \diamondsuit b = \left[\bigtriangledown \min_{i,j=1,2} (a_i \circ b_j),\ \bigtriangleup \max_{i,j=1,2} (a_i \circ b_j) \right].$$

These operations and the unary operation $-\boldsymbol{a}$ can be expressed by more explicit formulas as shown in Tables 14.1–14.4. 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	$[a_1, a_2] + [b_1, b_2] = [a_1 \mathbin{\overline{\bigtriangledown}} b_1,\ a_2 \mathbin{\overline{\bigtriangleup}} b_2].$
Subtraction	$[a_1, a_2] - [b_1, b_2] = [a_1 \mathbin{\overline{\bigtriangledown}} b_2,\ a_2 \mathbin{\overline{\bigtriangleup}} b_1].$

TABLE 14.1. Negation, addition, subtraction.

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

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

TABLE 14.2. Multiplication.

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

TABLE 14.3. Division by interval not containing 0.

Division, $0 \in \boldsymbol{b}$ $[a_1, a_2]/[b_1, b_2]$	$\boldsymbol{b} =$ $[0, 0]$	$[b_1, b_2]$ $b_1 < b_2 = 0$	$[b_1, b_2]$ $0 = b_1 < b_2$
$[a_1, a_2] = [0, 0]$	$\emptyset$	$[0, 0]$	$[0, 0]$
$a_1 < 0, a_2 \le 0$	$\emptyset$	$[a_2 \bigtriangledown b_1, +\infty)$	$(-\infty, a_2 \bigtriangleup b_2]$
$[a_1, a_2], a_1 < 0 < a_2$	$\emptyset$	$(-\infty, +\infty)$	$(-\infty, +\infty)$
$0 \le a_1, 0 < a_2$	$\emptyset$	$(-\infty, a_1 \bigtriangleup b_1]$	$[a_1 \bigtriangledown b_2, +\infty)$

TABLE 14.4. 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 14.3, 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.

14.7. 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.

14.8. Care needed with cancelMinus and cancelPlus. (informative)

14.8.1. *Comment by John Pryce.*

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 $\texttt{width}(\boldsymbol{x}) \geq \texttt{width}(\boldsymbol{y})$. 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.

14.8.2. *Numerical difficulties.*

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

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

15. Level 4 description

⚠ Probably does not need to exist in this standard.

CHAPTER 3

Kaucher Intervals

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

ANNEX A

Details of flavor-independent requirements

A.1. List of required functions

TBW

A.2. List of recommended functions

TBW

ANNEX B

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 C

Reproducibility

To be written. These fragments are moved from earlier in the hope they are useful here.

C.1. General arguments for reproducibility

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

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

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

C.2. A flavors example

Suppose the set-based and Kaucher flavors co-operate by sharing a type $\mathbb{T}$ whose intervals have lower and upper bounds that are `binary64` floating-point numbers. Suppose they implement some subset of $\mathbb{T}$'s operations in "tightest" mode, returning the smallest $\mathbb{T}$-interval that encloses the Level 1 result. This specification makes such operations flavor-independent (when acting on common intervals).

Then any common evaluation, that uses only this shared type and this subset of operations, gives identical results in both flavors, modulo the embedding map.

ANNEX D

Set-based flavor: decoration details and examples

D.1. Local decorations of arithmetic operations

D.1.1. Forward-mode elementary functions. For each of the required functions φ of §10.6, with the decoration scheme `com` > `dac` > `def` > `trv` > `ill` of Clause 11, Tables 1.1 to 1.2 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` and `abs`, together with `min` and `max` of any number of arguments.

The functions φ in Table 1.2 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 1.2 merely displays the set J for each function.

D.1.2. Interval case function.
⚠ JDP March 2013. This is probably wrong now. Subclause 10.6.4 defines the function `case`(c, g, h), and its required bare interval extension. It propagates decorations like other arithmetic operations, with local decoration d where

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

[*Note. Comparisons or overlap relations, as a mechanism for handling cases, are incompatible with the decoration concept since there is no way to account for exceptions. The* `case` *function handles decorations correctly. However, in most cases, functions defined using it give very suboptimal enclosures, and it is preferable to use methods illustrated in §11.9.*]

D.2. 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.

TABLE 1.1. Local decorations of required forward elementary functions. Normal mathematical notation is used to include or exclude an interval endpoint, e.g., $(-1, 1]$ denotes $\{\, x \in \mathbb{R} \mid -1 < x \leq 1 \,\}$. The specification for each function is written as a set of mutually exclusive cases.

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

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.

TABLE 1.2. Required forward elementary functions: step functions. The set of jump points is shown. The range and local decoration are computed by the algorithm below.

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

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

```
Input: nonempty x = [x̲, x̄], possibly unbounded.
y̲ = φ(x̲), ȳ = φ(x̄)
if y̲ = ȳ
  if x̲ ∉ J and x̄ ∉ J and x is bounded
    d = com
  else
    d = dac
  end if
else
  d = def
end if
Output: range enclosure [y̲, ȳ] and local decoration d.
```

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 $\mathrm{dec}(f, \boldsymbol{x}) = \texttt{trv}$ is not trivial: it asserts $p_{\texttt{emp}}$ and $p_{\texttt{def}}$ are both false. The first implies that $\boldsymbol{x}$ has a point in $\mathrm{Dom}(f)$, 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

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

   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 $du' = \mathrm{dec}(-, (\boldsymbol{y}, \boldsymbol{x})) = \texttt{dac}$, whence by (23) the decoration on $\boldsymbol{u}$ is $du = \min\{du', dy, dx\} = \min\{\texttt{dac}, \texttt{dac}, \texttt{dac}\} = \texttt{dac}$. Multiplication is also "nice", so the second operation similarly gives

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

   The constant 1, following §10.4.4, becomes a decorated interval function returning the constant value $[1, 1]_{\texttt{dac}}$. The next operation is again "nice", and gives

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

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

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

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

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

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

(iii) If $\boldsymbol{x} = [1,2]$, $\boldsymbol{y} = [1,1]$, the box $\boldsymbol{z}$ is now wholly outside $\mathrm{Dom}\, f$, 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.

D.3. Implementation of compressed interval arithmetic

Table 3.1 gives tables of compressed arithmetic, §11.11, for the four basic operations. Here c, d are bare decorations less than the threshold τ, and $\boldsymbol{x}, \boldsymbol{y}$ are bare intervals. Independently of τ, if any input is the decoration `ill` the result is `ill`, else if any input is the interval $\emptyset$ the result is $\emptyset$. The tables below give the remaining cases where

$$\mathtt{trv} \le c < \tau,\ \mathtt{trv} \le d < \tau, \text{ and } \boldsymbol{x},\ \boldsymbol{y} \text{ are nonempty.} \tag{40}$$

TABLE 3.1. Compressed interval operations for $+, -, \times, \div$ and $\sqrt{\cdot}$ with threshold $\tau \in \{\mathtt{trv}, \mathtt{def}, \mathtt{dac}, \mathtt{com}\}$.

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

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

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

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

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

Some examples of compressed arithmetic follow. In items (b) onwards, conditions (40) are assumed.

(a) Justification for $\texttt{emp} + \boldsymbol{x} = \texttt{emp}$, independent of τ.
This promotes to $(\emptyset, \texttt{emp}) + (\boldsymbol{x}, \tau) = \big(\emptyset, \min(\texttt{emp}, \tau, \texttt{emp})\big)$. Since $\texttt{emp} < \tau$ this equals $(\emptyset, \texttt{emp})$ which gives an exception (again because $\texttt{emp} < \tau$) so is recorded as the bare decoration emp. The same holds if $+$ is replaced by $-$, $\times$ or $\div$.

(b) Justification for $\boldsymbol{x} \times d = d$ independent of τ.
Since $\boldsymbol{x}$ is nonempty and $d \geq \texttt{trv}$, 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 $d \leq e$, so $\min(\tau, d, e) = d$.

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

(d) Justification for $\boldsymbol{x}/\boldsymbol{y}$ when $0 \in \boldsymbol{y} \neq [0, 0]$.
$\boldsymbol{x}/\boldsymbol{y}$ 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 $\tau > \texttt{trv}$ this gives an exception so the decoration trv is returned; if $\tau = \texttt{trv}$ it is not an exception, so the interval $\boldsymbol{x}/\boldsymbol{y}$ is returned.

(e) Justification for $\sqrt{\boldsymbol{x}}$ with $\boldsymbol{x} = [\underline{x}, \overline{x}]$ and $\underline{x} < 0 \leq \overline{x}$.
$\sqrt{\boldsymbol{x}}$ promotes to $\sqrt{(\boldsymbol{x}, \tau)}$, giving $(\sqrt{\boldsymbol{x}}, \texttt{trv})$ which in the given case equals $([0, \sqrt{\overline{x}}], \texttt{trv})$. As with the previous item, if $\tau > \texttt{trv}$ then trv is returned; if $\tau = \texttt{trv}$ then $\sqrt{\boldsymbol{x}}$ is returned.

D.4. The fundamental theorem of decorated interval arithmetic

We assume the seven-decoration set $\mathbb{D}$ of decorations defined by (15). However the proof of the fundamental theorem is largely independent of the particular set of decorations chosen.

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

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

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

We now prove:

Theorem D.4.1 (Fundamental Theorem of Interval Arithmetic, FTIA)**.**
Let $\boldsymbol{f}_{df} = f(\boldsymbol{x})$ be the result of evaluating an arithmetic expression $f(z_1, \ldots, z_n)$ over a bare box $\boldsymbol{x} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n) \in \overline{\mathbb{IR}}^n$ using any decorated interval version f of f. Then in addition to the enclosure

$$\boldsymbol{f} \supseteq \mathrm{Rge}(f \,|\, \boldsymbol{x}) \tag{41}$$

given by Moore's FTIA Theorem (page 16), we have

$$p_{df}(f, \boldsymbol{x}) \text{ holds.} \tag{42}$$

Proof. The case where $\boldsymbol{x}$ is empty is a special case. By case (Eval1) of the definition of a decorated interval version in §11.6, $\boldsymbol{f}_{df} = \emptyset_{\mathtt{ein}}$. Also $\mathrm{Rge}(f \,|\, \boldsymbol{x}) = \emptyset$ and by definition, $p_{\mathtt{ein}}(f, \boldsymbol{x})$ holds, so that (41, 42) hold.

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

The base case, where this number is zero, is that f is a variable, say z_i. Then it defines the function $f(x) = x_i$. By case (Eval2) in §11.6, $\boldsymbol{f} = \boldsymbol{x}_i = \mathrm{Rge}(f \,|\, \boldsymbol{x})$ and df is such that $p_{df}(\mathrm{Id}, \boldsymbol{x}_i)$ holds, where Id is the identity function $\mathrm{Id}(x) = x$ on $\mathbb{R}$ (specifically, $df = \mathtt{bnd}$ if $\boldsymbol{x}_i$ is bounded, and $df = \mathtt{dac}$ otherwise). Then $p_{df}(f, \boldsymbol{x}) = p_{df}(\mathrm{Id}, \boldsymbol{x}_i)$ holds. Thus (41, 42) hold.

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

By the inductive hypothesis the theorem holds for each g_i. (The case $k = 0$, where φ is a real constant or NaN, needs no special treatment.) So (41, 42) applied to g_i give for $i = 1, \ldots, k$

$$\boldsymbol{g}_i \supseteq \mathrm{Rge}(g_i \,|\, \boldsymbol{x}), \tag{43}$$

$$p_{dg_i}(g_i, \boldsymbol{x}) \quad \text{holds.} \tag{44}$$

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

$$\boldsymbol{f} \supseteq \mathrm{Rge}(\varphi \,|\, \boldsymbol{g}), \tag{45}$$

$$df = \min\{d\varphi, dg_1, \ldots, dg_k\} \tag{46}$$

for some $d\varphi$ such that

$$p_{d\varphi}(\varphi, \boldsymbol{g}) \text{ holds.} \tag{47}$$

We show first (41) and then (42). Denote here $u_k = g_i(x)$ for some $x \in \boldsymbol{x}$. Then

$$u_k = g_i(x) \in \mathrm{Rge}(g_i \,|\, \boldsymbol{x}) \subseteq \boldsymbol{g}_i. \tag{48}$$

For any $v \in \mathrm{Rge}(f \,|\, \boldsymbol{x})$, there is $x \in \boldsymbol{x}$ such that $v = f(x)$. Then, using (48, 45),

$$\begin{aligned} v = f(x) &= \varphi\big(g_1(x), \ldots, g_k(x)\big) \\ &= \varphi\big(u_1, \ldots, u_k\big) \\ &= \varphi(u) \in \mathrm{Rge}(\varphi \,|\, \boldsymbol{g}) \subseteq \boldsymbol{f}. \end{aligned}$$

Since v was arbitrary, this proves (41).

It remains to prove (42). Corresponding to the different meanings of the decorations, this is verified on a case by case basis, starting with the least decoration.

Case $df = \texttt{ill}$. Then either some $dg_i = \texttt{ill}$ or $d\varphi = \texttt{ill}$.

- If $dg_i = \texttt{ill}$, by (44) $\operatorname{Dom}\varphi$ is empty.
- If $d\varphi = \texttt{ill}$, hence by (47) $\operatorname{Dom}\varphi$ is empty.

In either case, by the definition of the point function f, $\operatorname{Dom} f$ is empty so (42) holds.

Case $df = \texttt{emp}$. Then either some $dg_i = \texttt{emp}$ or $d\varphi = \texttt{emp}$.

- If $dg_i = \texttt{emp}$, by (44) $\boldsymbol{x}$ is disjoint from $\operatorname{Dom} g_i$.
- If $d\varphi = \texttt{emp}$, by (47) $\boldsymbol{g}$ is disjoint from $\operatorname{Dom}\varphi$.

In either case there is no $x \in \boldsymbol{x}$ for which $f(x)$ is defined. That is, $\boldsymbol{x}$ is disjoint from $\operatorname{Dom} f$, and (42) holds.

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

Case $df = \texttt{def}$. Then each $dg_i \geq \texttt{def}$, and $d\varphi \geq \texttt{def}$. Thus by (44, 47), each g_i is everywhere defined on $\boldsymbol{x}$, with values in $\boldsymbol{g}_i$ by (43), and φ is everywhere defined on $\boldsymbol{g}$. Hence f is everywhere defined on $\boldsymbol{x}$ so again (42) holds.

Case $df = \texttt{dac}$. This is as the `def` case with the addition that the restriction of each g_i to $\boldsymbol{x}$ is everywhere defined and continuous, and the restriction of φ to $\boldsymbol{g}$ is everywhere defined and continuous. Hence the restriction of f to $\boldsymbol{x}$ is everywhere defined and continuous so again (42) holds.

Case $df = \texttt{bnd}$. Then each $dg_i \geq \texttt{bnd}$, and $d\varphi \geq \texttt{bnd}$. By similar reasoning, the restriction of f to $\boldsymbol{x}$ is everywhere defined, continuous and bounded so again (42) holds.
(In fact all one needs to deduce this is that each $dg_i \geq \texttt{dac}$, and $d\varphi \geq \texttt{bnd}$.)

Case $df = \texttt{ein}$. This cannot occur since $\boldsymbol{x}$ was assumed nonempty.

Hence all cases have been covered. This completes the induction step and the proof. □

D.5. Proofs of correctness for compressed interval arithmetic

⚠ To be completed.

ANNEX E

Further material for set-based standard (informative)

E.1. 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.

E.2. 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.

E.3. The "Not an Interval" object

⚠ TO BE REVISED

From §10.4.4, a real scalar function with no arguments—a mapping $\mathbb{R}^n \to \mathbb{R}^m$ with $n = 0$ and $m = 1$—is a **real constant**.

This specification of constants gives a Level 1 definition of NaN, "Not a Number"—not as a value, but as a constant function. $\mathbb{R}^0$ is the zero-dimensional vector space $\{0\}$—it has one element, conventionally named 0. The real numbers c are in one-to- one correspondence with the mappings $c() : 0 \mapsto c$, so that $\mathbb{R}$ can be identified with the *total* functions $\mathbb{R}^0 \to \mathbb{R}$. There is one *non-total* $c()$, the function NaN() with empty domain and, therefore, no value.

From the definition in §10.4.3, an interval extension of a real constant with value c is any zero-argument interval function that returns an interval containing c. The *natural extension* returns the interval $[c, c]$.

Its natural interval extension is the constant interval function whose value is the empty interval.

the zero-argument function with empty domain is the real constant function with value NaN, "Not a Number". It is easily seen that NaN's natural interval extension is the interval constant function with value $\emptyset$, and its natural decorated interval extension is the decorated interval constant function with value 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.

ANNEX F

Level 2 extra bits

F.1. Rationale for defined hulls and text representation

⚠ This looks like useful commentary on design matters, but it is unclear how appropriate it is for the final text.

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

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

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

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

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

So let's tie things down. What, and how far? I think requiring implementors to define an unambiguous, platform-independent hull operation for all interval types is not too much to ask.

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).