P1788: IEEE Standard For Interval Arithmetic Version 04.1

John Pryce and Christian Keil, Technical Editors

6. Level 2 description

Objects and operations at Level 2 are said to have **finite precision**. They are the entities from which implementable interval algorithms may be constructed. Level 2 objects are called **datums**[9]Since the standard deals with numeric functions of intervals (such as the midpoint) and interval functions of numbers (such as the construction of an interval from its lower and upper bounds), this clause involves both numeric and interval datums, as well as the unique set $\mathbb{D}$ of **decorations**.

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

– **Bare interval types**, see §6.3, representing finite sets of (mathematical, Level 1) intervals.
– **Decorated interval types**, see §**??**, representing finite sets of decorated intervals.
– **Compressed interval types**, see §**??**, which implement compressed interval arithmetic.

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

This standard uses the term **operation of a type** $\mathbb{T}$, where $\mathbb{T}$ is a bare or decorated interval type, to mean any of the following:

(a) A $\mathbb{T}$-interval extension (§6.5) of one of the required or recommended operations of §6.6, whose inputs and output are $\mathbb{T}$-intervals.
(b) A function such as the midpoint, whose input is a $\mathbb{T}$-interval and output is a numeric value.
(c) A constructor, whose input is numeric or text and output is a $\mathbb{T}$-interval.
(d) The $\mathbb{T}$-interval hull, regarded as a conversion operation, see §6.4.2.

Such an operation is also called a **$\mathbb{T}$-version** of the corresponding mathematical point operation (e.g., $\mathbb{T}$-version of $+$), or generic operation of that name (e.g., $\mathbb{T}$-version of the midpoint function).

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

6.1. Datums are tagged by names. A format or type is an abstraction of a particular way to represent numbers or intervals—e.g., "IEEE 64 bit binary" for numbers—focusing on the Level 1 objects represented, and hiding the Level 3 method by which it is done.

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

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

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

where the name is some symbol that uniquely identifies the format or type. The Level 1 value is said to be **tagged** by the name. This achieves two needed properties: distinct formats or types are disjoint sets; and two datums are equal if and only if they represent the same Level 1 value tagged by the same name.

Names are omitted 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.*]

6.2. Number formats. Having regard to §6.1, a **number format**, or just format, is the set of all pairs (x, f) where x belongs to a finite subset $\mathbb{F}$ of the extended reals $\overline{\mathbb{R}}^*$ that contains $-\infty$ and $+\infty$, and f is a name for $\mathbb{F}$. Normally the format is identified with the set $\mathbb{F}$. A floating-point format in the 754 sense, such as `binary64`, is identified with the number format where $\mathbb{F}$ comprises the set of extended-real numbers that are exactly representable in that format, where –0 and +0 both represent the mathematical number 0.

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

6.3. Bare interval types.

6.3.1. *Definition.* A **bare interval type** is an arbitrary finite set $\mathbb{T}$ of pairs $(\boldsymbol{x}, t)$ such that (Empty, t) and (Entire, t) are members of $\mathbb{T}$, where t is a unique name for $\mathbb{T}$.

With this simplification, a bare interval type is an arbitrary finite set $\mathbb{T}$ of intervals that includes Empty and Entire.

An interval in $\mathbb{T}$ may be called a **$\mathbb{T}$-interval**; a box with $\mathbb{T}$-interval components may be called a **$\mathbb{T}$-box**.

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

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

6.3.2. *Inf-sup and mid-rad types.* The **inf-sup type derived from** a given number format $\mathbb{F}$ (called the **inf-sup $\mathbb{F}$ type** for short) is the bare interval type $\mathbb{T}$ comprising all intervals whose endpoints are in $\mathbb{F}$, together with Empty. Note that Entire is in $\mathbb{T}$ because $\pm\infty \in \mathbb{F}$ by the definition of a number format, so $\mathbb{T}$ satisfies the requirements for a bare interval type given in §6.3.1.

A 754-conforming implementation shall support the inf-sup type derived from the basic 754 format `binary64`, and may support the inf-sup types of any of the other basic formats `binary32`, `binary128`, `decimal64`, and `decimal128`.

Mid-rad types are not specified by this standard but are useful for examples. A mid-rad bare interval type is taken to be one whose nonempty bounded intervals comprise all intervals of the form $[m - r, m + r]$, where m is in some number format $\mathbb{F}$, and r is in a possibly different number format $\mathbb{F}'$, with m, r finite and $r \geq 0$. From the definition in §6.3.1 the type must also contain Empty and Entire; it may also contain semi-infinite intervals.

6.3.3. *Multi-precision interval types.* Multi-precision floating point systems 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 used to define a corresponding infinite sequence of interval types $\mathbb{T}_n$.

For those 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 §6.4.

It is necessary that these define an *infinite sequence* of interval types. It is not possible to take the (infinite) set, of all $\mathbb{T}_n$-intervals for arbitrary n, as a *single* type, because it 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 multi-precision interval systems that conform to this standard.

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

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

$$\boldsymbol{y} = \text{hull}(\boldsymbol{s}),$$

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

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

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

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

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

[*Examples. It is easy to see that 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 (§6.3.2) where $\mathbb{F}$ and $\mathbb{F}'$ are the IEEE binary64 format, and let $\boldsymbol{s}$ be the interval $[-1, 1+\epsilon]$ where $1+\epsilon$ is the next binary64 number above 1. Clearly any minimal interval (m, r) enclosing $\boldsymbol{s}$ has $r = 1+\epsilon$. But m can be any of the many binary64 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. For more exotic $\boldsymbol{s}$ it could be expensive.]

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

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

6.4.2. *Interval conversion.* An implementation shall provide, for each supported bare interval type $\mathbb{T}$, an operation that returns $\text{hull}_{\mathbb{T}}(\boldsymbol{x})$ (as an interval of type $\mathbb{T}$) for any interval $\boldsymbol{x}$ of any supported bare interval type.

6.4.3. *Hull in several dimensions.* In n dimensions the $\mathbb{T}$-hull is defined componentwise, namely the hull of an arbitrary subset $\boldsymbol{s}$ of $\mathbb{R}^n$ is $\boldsymbol{y} = (\boldsymbol{y}_1, \ldots, \boldsymbol{y}_n)$ where

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

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 $\boldsymbol{s}$.

6.5. Level 2 interval extensions. Given a bare interval type $\mathbb{T}$, a **$\mathbb{T}$-interval extension** of an n-variable scalar point function 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

$$\boldsymbol{f}(\boldsymbol{x}) \supseteq \text{Range}(f \,|\, \boldsymbol{x}). \tag{20}$$

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 essentially equivalent to a full Level 1 extension of f, see §5.4.3, namely $\boldsymbol{f}^*$ where

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

for any subset $\boldsymbol{s}$ of $\mathbb{R}^n$. Then $\boldsymbol{f}^*(\boldsymbol{s}) = \boldsymbol{f}(\boldsymbol{s})$ for $\boldsymbol{s} \in \mathbb{T}^n$, so this document does not distinguish between the Level 2 and Level 1 extensions..

6.6. Operations on bare intervals.

6.6.1. *Accuracy modes for inf-sup types.* The standard defines **accuracy modes** that indicate how near an operation is to being as tight as possible.

[*Note. These modes are specified for inf-sup types only. "Tightest" and "valid" clearly apply to any interval type, but there seems no simple way to define an analogue of "accurate" for general types.*]

For a given number format $\mathbb{F}$ and an extended-real number x, $\mathtt{nextUp}(x)$ is defined to be $+\infty$ if $x = +\infty$, and the least member of $\mathbb{F}$ greater than x otherwise; $\mathtt{nextDown}(x)$ is defined to be $-\infty$ if $x = -\infty$, and the greatest member of $\mathbb{F}$ less than x otherwise.

Given an interval $\boldsymbol{x} = [\underline{x}, \overline{x}]$ of the inf-sup type $\mathbb{T}$ derived from $\mathbb{F}$, $\mathtt{widen}(\boldsymbol{x})$ is the $\mathbb{T}$-interval defined by

$$\mathtt{widen}(\boldsymbol{x}) = [\mathtt{nextDown}(\underline{x}), \mathtt{nextUp}(\overline{x})).$$

[*Note. That is,* `widen` *moves each finite endpoint of $\boldsymbol{x}$ outward to the next $\mathbb{F}$-number. For 754 formats and others based on a radix-significand-exponent form, this is often called a change of one* ulp *(unit in the last place).*]

For a $\mathbb{T}$-box $\boldsymbol{x} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$, this function acts componentwise to produce the $\mathbb{T}$-box

$$\mathtt{widen}(\boldsymbol{x}) = (\mathtt{widen}(\boldsymbol{x}_1), \ldots, \mathtt{widen}(\boldsymbol{x}_n)).$$

For a $\mathbb{T}$-interval extension $\boldsymbol{f}$ of an n-variable scalar point function f, the standard specifies three **accuracy modes** for $\boldsymbol{f}$:

Tightest: $\boldsymbol{f}(\boldsymbol{x})$ shall equal the $\mathbb{T}$-hull of $(\text{Range}(f \,|\, \boldsymbol{x}))$, for any $\mathbb{T}$-box $\boldsymbol{x}$. That is, equality is to hold in (20).

Forward		Reverse	
Name	Accuracy	Name	Accuracy
`add`(x, y)	tightest	`sqrRev`(x)	accurate
`sub`(x, y)	tightest	`invRev`(x)	accurate
`mul`(x, y)	tightest	`absRev`(x)	accurate
`div`(x, y)	tightest	`pownRev`(x, p)	accurate
`inv`(x)	tightest	`sinRev`(x)	accurate
`sqrt`(x)	tightest	`cosRev`(x)	accurate
`hypot`(x, y)	tightest	`tanRev`(x)	accurate
`case`(b, g, h)	tightest	`coshRev`(x)	accurate
`sqr`(x)	tightest		
`pown`(x, p)	accurate		
`pow`(x, y)	accurate		
`exp,exp2,exp10`(x)	tightest		
`log,log2,log10`(x)	tightest		
`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		

TABLE 8. Accuracy levels for required arithmetic operations.

Accurate: $\boldsymbol{f}(\boldsymbol{x})$ shall be contained in the $\mathbb{T}$-hull of Range$(f \mid \texttt{widen}(\boldsymbol{x}))$, for any $\mathbb{T}$-box $\boldsymbol{x}$. That is, the result lies between the outward-rounded lower and upper bounds of the exact range of a slightly expanded input box.

Valid: No requirement beyond (20).

6.6.2. *Arithmetic operations.* An implementation shall provide a $\mathbb{T}$-interval extension of each forward and reverse arithmetic operation in §5.6.1, 5.6.2, 5.6.3, for each supported bare interval type $\mathbb{T}$ (and hence for its derived decorated type, see §6.7). [*Note. For operations with some integer arguments, such as integer power x^n, only the real arguments are replaced by intervals.*]

For a 754-conforming type, each required operation shall have a version of that type with accuracy mode as in Table 8. For recommended operations, and for any type that is not 754-conforming, the accuracy mode is implementation-defined.

An implementation may provide more than one version of some operations for a given type. For instance it may provide an "accurate" version of some operation in addition to a required "tightest" one, to offer a trade-off between accuracy and speed.

6.6.3. *Non-arithmetic operations.* An implementation shall provide $\mathbb{T}$-interval versions of each operation in §5.6.4, for each supported bare interval type $\mathbb{T}$ (and hence for its derived decorated type, see §6.7).

These operations shall return the $\mathbb{T}$-interval hull of the exact result. In particular, for any inf-sup type, both `intersection` and `convexHull` shall return the exact result.

6.6.4. *Constructors and conversions.*

⚠ This is provisional and is matched to the corresponding Level 1 operations.

An implementation shall provide the following constructor operations. *typeOf* indicates that the name of the operation specifies the destination interval type, which may be unrelated the operands' number formats. E.g., *typeOf*-`num2interval` might stand for any of `binary32num2interval`, `decimal64num2interval`, etc. There shall be a version of such operations for each supported interval type.

typeOf-`nums2interval`(l, u). If the numbers represented by the floating point datums l, u satisfy the conditions in §5.2 for defining a nonempty interval $\boldsymbol{x} = [l, u]$—i.e., $-\infty \leq l \leq u \leq +\infty$, $l < +\infty$, $u > -\infty$—the constructor is said to *succeed*, and returns the tightest interval of the destination type containing $\boldsymbol{x}$. Otherwise it is said to *fail*, and returns Empty.

The types of l and u may be any supported number formats and need not be the same.

typeOf-`num2interval`(x) has the same effect as *typeOf*-`nums2interval`(x, x). In particular it succeeds if and only if the floating point datum x is a finite number.

typeOf-`Entire`$()$ succeeds, and has the same effect as *typeOf*-`nums2interval`$(-\infty, +\infty)$.

typeOf-`Empty`$()$ succeeds, and returns the empty interval of the destination type.

typeOf-`text2interval`(s) If the text string s defines a mathematical interval $\boldsymbol{x}$, this constructor *succeeds*, and returns the tightest interval of the destination type containing $\boldsymbol{x}$. Otherwise it *fails*, and returns the empty interval.

⚠ A motion is needed to decide the details. Vienna§6 has a specification we may like to follow. Meantime I assume, in examples, that things like `text2interval`("[1.2, 3.4]") and `text2interval`("Empty") have the natural effect.

6.6.5. *Reverse-mode elementary functions.*

⚠ The list from a revised Motion 11 will go here.

6.6.6. *Comparison operations.*

⚠ No motion about these yet.

6.6.7. *Other operations.*

⚠ I expect intersection, hull, midpoint, etc., will go here.

It needs to be decided how such non-arithmetic operations handle decorations. Also how decorations affect comparison functions:

⚠ We need to specify how the interval part of a `ill` interval behaves w.r.t. real-valued functions like radius (always NaN?) and "forget decoration" (gives Empty?).

6.6.8. *Reduction operations.* Implementations shall provide an exact dot product operation.

⚠ Details TBW

6.6.9. *Recommended operations (informative).*

6.6.10. *Accuracy of recommended elementary functions.*

⚠ This will give Vincent Lefevre's list of accuracy info for the functions listed in Level 1.

6.6.11. *Reverse-mode elementary functions.*

⚠ The list from a revised Motion 11 will go here.

6.6.12. *Other operations.*

⚠ The list from a revised Motion 11 will go here.

6.7. Decorated interval types.

Given a bare interval type $\mathbb{T}$, the derived decorated interval type $\mathbb{DT}$ consists of all decorated intervals whose bare interval part is a $\mathbb{T}$-interval, tagged with the name of $\mathbb{T}$. That is, a member of $\mathbb{DT}$ can be regarded as a triple $(\boldsymbol{x}, d, t)$ where $(\boldsymbol{x}, t)$ is in $\mathbb{T}$ and d is a valid decoration for $\boldsymbol{x}$ according to (**??**).

6.8. Representations.

[*Note. Some of these definitions appeared in previous versions of this paper or in the draft standard, others didn't.*]

- A *representation* of an interval type comprises a set $\overline{\mathbb{IF}}$ called an i-format, and whose members are called (level 3) *interval objects*, together with a map r from a subset of $\overline{\mathbb{IF}}$ (the "valid" objects) to $\mathbb{T}$. If object X maps to datum $\boldsymbol{x}$, we say X represents $\boldsymbol{x}$. (DS§3.1 Table 1 note *b*: "Not every interval object necessarily represents an interval datum, but when it does, that datum is unique. Each interval datum has at least one representation, and may have more than one.")
  [*Note. A representation is not an approximation—it means just what it says. E.g., an interval object in mid-rad form with midpoint* $m = 1$ *and radius* $r = 1\text{e}{-300}$ *means precisely the mathematical interval* $[1 - 10^{-300}, 1 + 10^{-300}]$.]
- A *text representation* of an interval type is a representation whose i-format is a set $\overline{\mathbb{IT}}$ of text strings. [*Note. This is related to the Motion 17 notion of a ti-format. The latter assumes an inf-sup representation, while the present definition does not.*]
- A representation is *standardized* if there is also provided a map s from the whole of $\mathbb{T}$ into $\overline{\mathbb{IF}}$, such that $s(\boldsymbol{x})$ is a representation of $\boldsymbol{x}$:

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

  The object $s(\boldsymbol{x})$ is called the "standard representation" of the datum $\boldsymbol{x}$.
  [*Example. In an inf-sup representation,* r *might map both the objects* `(-0,3)` *and* `(+0,3)` *to the interval datum* $[0, 3]$. *Suppose the standardized representation always uses* `+0`*; then* s *would map* $[0, 3]$ *to* `(+0,3)`.]
  [*Note. A* standardized text representation *of* $\mathbb{T}$ *essentially defines a way to write any* $\mathbb{T}$*-interval out in text form and read it back* exactly*. Think of map* s *as "write" and* r *as "read". See §6.9.2.*]

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

6.9.1. *The hull.*

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

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

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

6.9.3. *Reproducibility.*

6.9.4. *Interval formats.*

6.9.5. $\mathbb{F}$*-intervals.*

6.9.6. *Finite precision hull.*

6.9.7. *Interval format conversion.*

6.10. Decoration system. ⚠ To be written.

6.10.1. *Implementation considerations.* (Informative.)

6.10.2. *Forgetful operations.* Implementations shall provide operations

intervalPart($\boldsymbol{x}$)
decorationPart($\boldsymbol{x}$)

that return the interval or decoration part of the input, respectively.