8.8. The decoration system.

8.8.1. *Decorations and decorated intervals.* A decoration is information attached to an interval; the combination is called a decorated interval. There are two main aims of interval calculations:

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

Traditional interval analysis targets the first aim; decorated intervals target the second.

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

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

– call the `newDec` operation, see §8.8.4, 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.

Decorations are based on the desire that, from an interval evaluation of a real function f at 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.

This applies to *arithmetic* interval operations. Non-arithmetic operations, such as hull and intersection of two intervals, occur in too varied contexts to allow a uniform decoration-handling scheme. Hence they ignore decorations, and give undecorated output. Users are responsible for the appropriate propagation of decorations by these operations.

As in §8.5, 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 B gives formal proofs in terms of a particular definition, and indicates how this relates to expressions in some programming languages.

To make the decoration system available, implementations need only provide a decorated version (§8.8.4) of each Level 2 interval extension of an arithmetic operation, and also some auxiliary functions.

The P1788 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.

The system is defined here at a mathematical level, with the finite-precision aspects in §9.13. §8.8.2 to §8.8.3 give the basic concepts. §8.8.4 defines how decorations are bound to library interval operations. §8.8.6 discusses the decoration of user-defined arithmetic operations. §8.8.8 defines a restricted decorated interval arithmetic that suffices for some important applications and is easier to implement efficiently.

In Annex B, extensive examples of the meaning and use of decorations are in Clause 14; and Clause 17 contains a rigorous theoretical foundation, including a proof of the Fundamental Theorem of Decorated Interval Arithmetic.

8.8.2. *Definitions and basic properties.* Formally, a decoration d is a property $p_d(f,\boldsymbol{x})$ 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 and more usually, d is identified with the set of pairs for which the property holds:

$$d = \{\, (f, \boldsymbol{x}) \mid p_d(f, \boldsymbol{x}) \text{ is true} \,\}. \tag{12}$$

This standard provides a set $\mathbb{D}$ of either five or six decorations. The basic five are:

Value	Short description	Property	Definition
`dac`	defined & continuous	$p_{\texttt{dac}}(f, x)$	$\boldsymbol{x}$ is a nonempty subset of $\operatorname{Dom}(f)$, and the restriction of f to $\boldsymbol{x}$ is continuous;
`def`	defined	$p_{\texttt{def}}(f, \boldsymbol{x})$	$\boldsymbol{x}$ is a nonempty subset of $\operatorname{Dom}(f)$;
`trv`	trivial	$p_{\texttt{trv}}(f, \boldsymbol{x})$	always true (so gives no information);
`emp`	empty	$p_{\texttt{emp}}(f, \boldsymbol{x})$	$\boldsymbol{x} \cap \operatorname{Dom}(f)$ is empty;
`ill`	ill-formed	$p_{\texttt{ill}}(f, \boldsymbol{x})$	$\operatorname{Dom}(f)$ is empty.

(13)

An implementation may, and one that supports more than one flavor shall, provide the sixth decoration `com`, see §8.8.7.

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

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

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

Treating the decorations as sets as in (12), `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. (14)

Namely the definitions (13) give:

$$\texttt{dac} \subseteq \texttt{def} \subseteq \texttt{trv} \supseteq \texttt{emp} \supseteq \texttt{ill}, \qquad \text{note the change from } \subseteq \text{ to } \supseteq; \tag{15}$$

$$\texttt{dac} \text{ and } \texttt{def} \text{ are disjoint from } \texttt{emp} \text{ and } \texttt{ill}. \tag{16}$$

Hence for any $(f, \boldsymbol{x})$ there is a unique smallest (in the containment order (15)), decoration such that $p_d(f, \boldsymbol{x})$ is true, called the **strongest decoration of** $(f, \boldsymbol{x})$, or of f over $\boldsymbol{x}$, and written $\operatorname{dec}(f, \boldsymbol{x})$. That is:

$$\operatorname{dec}(f, \boldsymbol{x}) = d \iff p_d(f, \boldsymbol{x}) \text{ holds, but } p_e(f, \boldsymbol{x}) \text{ fails for all } e \subset d. \tag{17}$$

8.8.3. *Permitted combinations.*

Ill-formed intervals: The "ill-formed" decoration `ill` propagates unconditionally through arithmetic expressions. Namely, the decorated interval result of a library arithmetic operation is ill-formed (decorated with `ill`) if and only if one of its inputs is ill-formed.

Ill-formed decorated intervals result from an invalid constructor call, and also may be produced in any context where an implementation determines, statically or dynamically, that it is to do interval-evaluation of an expression that, as a point function, is nowhere-defined. They are called NaI, **Not an Interval**. Generally, an implementation behaves as if there is only one NaI, whose interval part is empty, i.e. NaI $= \emptyset_{\texttt{ill}}$. An exception is that other information may be stored in an NaI, in an implementation-defined way, and functions may be provided for a user to interrogate this for diagnostic purposes. [*Note. An example of* NaI *would be an invalid constant expression such as* $[1, 1]/[0, 0]$, *which comes from the undefined point constant* $1/0$. *With more analysis, a compiler might determine that evaluating* $\sqrt{-[1, 1] - \boldsymbol{x}^2}$ *should give* NaI, *for any input interval* $\boldsymbol{x}$, *because* $\sqrt{-1 - x^2}$ *is everywhere undefined as a real point function.*]

Decorations of Empty: By design, a decorated interval $\boldsymbol{y}_{dy}$ is always such that $\boldsymbol{y} \supseteq \operatorname{Rge}(f \mid \boldsymbol{x})$ and $p_{dy}(f, \boldsymbol{x})$ holds for some pair $(f, \boldsymbol{x})$. If $dy =$ `dac` or `def` then by definition $\boldsymbol{x}$ is nonempty and f is everywhere defined on it, so that $\operatorname{Rge}(f \mid \boldsymbol{x})$ is nonempty. Hence the decorated intervals $\emptyset_{\texttt{dac}}$ and $\emptyset_{\texttt{def}}$ are contradictory: implementations shall not produce them.

No other combinations are essentially forbidden. However it is a consequence of a Level 2 requirement that a conforming interval arithmetic operation never gives the empty set any decoration but `emp` or `ill`; thus $\emptyset_{\texttt{trv}}$ does not occur in normal evaluation of expressions.

Unbounded `dac` intervals: An unbounded interval may be decorated `dac`. At Level 1 there is little reason for this to happen, but at Level 2 it indicates that overflow occurred. The decoration system handles this consistently. E.g., if evaluating $\exp(x)$ on $[0, 1000]$ gives $[1, +\infty]_{\texttt{dac}}$, then the implementation determined that $\exp()$ is continuous on $[0, 1000]$, but e^{1000}, the upper bound of the range, is too large to be representable.

8.8.4. *Decorations and arithmetic operations.* Correct use of decorations when evaluating an expression has two parts: correctly initialize the input intervals; and evaluate using decorated interval extensions of library operations.

To provide correct initialization the function `newDec`() is provided. For a single bare interval $\boldsymbol{x}$, `newDec`() decorates it with $\mathrm{dec}(\mathrm{Id}, \boldsymbol{x})$, the strongest decoration dx that makes $p_{dx}(\mathrm{Id}, \boldsymbol{x})$ true, where Id is the identity function $\mathrm{Id}(x) = x$ for $x \in \mathbb{R}$. That is,

$$\texttt{newDec}(\boldsymbol{x}) = \boldsymbol{x}_d \qquad \text{where} \qquad d = \mathrm{dec}(\mathrm{Id}, \boldsymbol{x}) = \begin{cases} \texttt{dac} & \text{if } \boldsymbol{x} \text{ is nonempty,} \\ \texttt{emp} & \text{if } \boldsymbol{x} \text{ is empty.} \end{cases} \tag{18}$$

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

$$\texttt{newDec}(\boldsymbol{x}_{dx}) = \texttt{newDec}(\boldsymbol{x}). \tag{19}$$

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

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 is also called a **decorated version** of that bare interval extension. 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.

The definition of a bare interval extension implies

$$\boldsymbol{w} \supseteq \mathrm{Rge}(\varphi \,|\, \boldsymbol{v}), \qquad \text{(enclosure).} \tag{20}$$

φ determines a dv_0 such that

$$p_{dv_0}(\varphi, \boldsymbol{v}) \text{ holds,} \qquad \text{(a “local decoration”).} \tag{21}$$

It then evaluates the output decoration dw by

$$dw = \min\{dv_0, dv_1, \ldots, dv_k\}, \qquad \text{(the “min-rule”),} \tag{22}$$

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

$$\texttt{dac} > \texttt{def} > \texttt{trv} > \texttt{emp} > \texttt{ill}. \tag{23}$$

[*Notes.*

1. *Let* $\mathsf{f}(z_1, \ldots, z_n)$ *be an expression defining a real point function* $f(x_1, \ldots, x_n)$. *Then decorated interval evaluation of* f *on a correctly initialized input decorated box* $\boldsymbol{x}_{dx}$ *gives a decorated interval* $\boldsymbol{y}_{dy}$ *such that not only, by the Fundamental Theorem of Interval Arithmetic, one has*

$$\boldsymbol{y} \supseteq \mathrm{Rge}(f \,|\, \boldsymbol{x}) \tag{24}$$

but also

$$p_{dy}(f, \boldsymbol{x}) \text{ holds.} \tag{25}$$

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

Briefly, (18) 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 (22) embodies basic facts of set theory and analysis, such as "If each of a set of functions is everywhere defined [resp. continuous], their composition has the same property" and "If any of a set of functions is nowhere defined, their composition has the same property". It causes correct propagation of decorations through each arithmetic operation, and hence through a whole expression.

2. *In the same way as the enclosure requirement (20) 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 (21). The ideal choice is the strongest decoration d such that $p_d(\varphi, \boldsymbol{v})$ holds, that is to take*

$$dv_0 = \mathsf{dec}(\varphi, \boldsymbol{v}). \tag{26}$$

This is easily computable in finite precision for the arithmetic operations in §8.6, 8.7—see the tables in Annex B, Clause 13. However, functions may be added to the library in future for which (26) is impractical to compute for some arguments $\boldsymbol{v}$. Hence the weaker requirement (21) is made.

]

8.8.5. *Operations that do not propagate decorations.*

The following are not interval extensions of point functions:

– The reverse-mode operations of §8.6.4.
– The cancellative operations `cancelPlus`$(\boldsymbol{x}, \boldsymbol{y})$ and `cancelMinus`$(\boldsymbol{x}, \boldsymbol{y})$ of §8.6.5.
– The set-oriented operations `intersection`$(\boldsymbol{x}, \boldsymbol{y})$ and `convexHull`$(\boldsymbol{x}, \boldsymbol{y})$ of §8.6.6.
– The numeric functions of §8.6.8; the comparisons and other boolean-valued functions of §8.6.9; and the overlap function of §8.7.2.

The decorated interval version of each such operation takes decorated interval inputs and gives the result obtained by discarding the decorations and applying the corresponding bare interval operation. Users are responsible for decorating this result, where relevant, as may be appropriate for an application.

An implementation may provide versions of these operations that compute a bare interval result as above, and add a decoration suited to a particular application. How this is done is language-defined.

8.8.6. *User-supplied functions.* A user may define a decorated version of some point function f, to be used within expressions as if it were a library function. Correct decoration results are obtained, provided it is a decorated interval extension of f as defined in §8.8.4.

[*Examples.*

(1) *In an application, an interval extension of the function*

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

was required. As it stands it gives poor 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 $\mathsf{Rge}(f \mid \boldsymbol{x}) = [2, 2\frac{1}{2}]$.

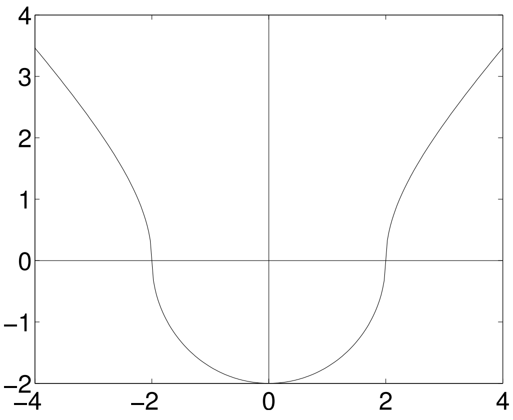
Thus it is useful to code a tight interval extension by special methods, e.g. monotonicity arguments, and provide this as a new library function. Suppose this has been done. To convert it to a decorated interval extension just entails adding code to provide a local decoration and combine this with the input decoration by the min-rule (22). In this case it is straightforward to compute the strongest local decoration $d = \mathsf{dec}(f, \boldsymbol{x})$, as follows.

$$d = \begin{cases} \texttt{emp} & \textit{if } \boldsymbol{x} = \emptyset \textit{ or } \boldsymbol{x} = [0, 0], \\ \texttt{trv} & \textit{if } 0 \in \boldsymbol{x} \neq [0, 0], \\ \texttt{dac} & \textit{if } 0 \notin \boldsymbol{x} \neq \emptyset. \end{cases}$$

(2) *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*

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

see the diagram.



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

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

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

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

]

8.8.7. *Multi-flavor implementations.* A multi-flavor implementation shall, and other implementations may, provide the sixth decoration `com`, defined in §6.3, namely:

Value	Short description	Property	Definition
`com`	common	$p_{\texttt{com}}(f, \boldsymbol{x})$	$\boldsymbol{x}$ is a bounded, nonempty subset of $\mathrm{Dom}(f)$; f is continuous at each point of $\boldsymbol{x}$; and the computed interval $f(\boldsymbol{x})$ is bounded.

(27)

Its position in the containment and propagation orders is given by

$$\texttt{com} \subseteq \texttt{dac} \subseteq \texttt{def} \subseteq \texttt{trv} \supseteq \texttt{emp} \supseteq \texttt{ill},$$

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

The `newDec` function is amended to give the decoration `com` instaed of `dac` to a bounded, nonempty interval.

Note `com` describes a Level 2 property. When f is a library arithmetic operation φ, the computed interval $\varphi(\boldsymbol{x})$ means the enclosure of $\mathrm{Rge}(\varphi \,|\, \boldsymbol{x})$ computed by a particular Level 2 interval extension of φ, giving a result of some interval type $\mathbb{T}$. Thus $p_{\texttt{com}}(\varphi, \boldsymbol{x})$ indicates a finite-precision evaluation that is common in the flavor sense, giving a bounded enclosure of the range.

When f is a function defined by an expression, the computed interval $f(\boldsymbol{x})$ means the enclosure $\boldsymbol{y}$ of $\mathrm{Rge}(f \,|\, \boldsymbol{x})$ produced by interval evaluation of the expression at Level 2. Evaluation need not use just one interval type $\mathbb{T}$: any combination of supported types is permitted. If the final $\boldsymbol{y}$ is decorated with `com`, it follows from the propagation rules that the whole evaluation was common. That is, the final enclosure of the range is bounded, as were all the intermediate intervals. In addition, the function f is everywhere continuous on the bounded, nonempty input box $\boldsymbol{x}$.

8.8.8. *Compressed arithmetic with a threshold (informative).*

⚠ Some of this needs to be moved to Level 2.

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** derived from its **parent type** $\mathbb{T}$, with its own objects and library of operations. Conversions are provided between a compressed type and its parent type.

Compressed arithmetic uses the standard set of 5 or 6 decorations (13). The context is that, frequently, the use that is made of a decorated interval function 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 (23):

$dy \geq \tau$ 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.

For such uses, one needs to store an interval's value, or its decoration, but never both at once. A compressed interval is an object whose value is either an arbitrary bare interval (of the parent type), or an arbitrary bare decoration, with the exception that the empty interval is not used: the decoration `emp` is used instead.

Since, for any practical interval type $\mathbb{T}$, a decoration fits into less space than an interval, one can implement arithmetic on "compressed interval" objects 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.

The threshold has the properties of an attribute logically associated with a program block, as in 754-2008 §4 (see also 754-2008 §2.1.7). It shall be possible to specify a constant value for it as in 754-2008 §4.1; dynamic mode specification should be supported as in 754-2008 §4.2.

A compressed interval constructor behaves as if it first constructs a decorated interval $\boldsymbol{x}_{dx}$. If $dx < \tau$, the result is the bare decoration dx; otherwise the bare interval $\boldsymbol{x}$. In particular an invalid construction, and valid construction of an empty interval, always produce compressed intervals holding the decorations `ill` and `emp` respectively, irrespective of τ.

An implementation shall provide the enquiry function `isInterval`$(\boldsymbol{x})$ which returns true if the compressed interval $\boldsymbol{x}$ is an interval, false if it is a decoration.

An implementation shall provide the function `normalInterval`$(\boldsymbol{x})$ which compressed interval $\boldsymbol{x}$ is an interval, false if it is a decoration.

Arithmetic operations on compressed intervals derive from normal decorated interval operations. The behavior depends on the threshold, which the user, or potentially the implementation, can choose to fit the use made of the result. The results are determined by **worst case semantics** rules that treat a bare decoration as representing a set of decorated intervals. These follow necessarily if the fundamental theorem is to remain valid. 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.

(a) 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 $\boldsymbol{y}$.

(b) For arithmetic operations with at least one bare decoration input, the result is always a bare decoration. A bare decoration d in $\{\texttt{emp}, \texttt{ill}\}$ is treated as $\emptyset_d$. A bare decoration d in $\{\texttt{trv}, \texttt{def}, \texttt{dac}\}$ is treated (conceptually, not algorithmically) as an arbitrary $\boldsymbol{x}_d$ with nonempty interval $\boldsymbol{x}$ that is compatible with d: for d in $\{\texttt{trv}, \texttt{def}\}$, $\boldsymbol{x}$ is unrestricted, while for $d = \texttt{dac}$ it is bounded. A bare interval is treated as in item (a). 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 (15)) enclosing all resulting dy is returned.

Since there are only a few decorations, one can prepare complete operation tables according to these rules, and only these tables need to be implemented. Sample tables for a number of operations are given in §16 in Annex B, together with some worked examples of compressed arithmetic.

If compressed arithmetic is implemented, it shall provide versions of all the required operations of §8.6, and should provide the recommended operations of §8.7.

⚠ It needs to be decided how numeric functions such as midpoint work on a compressed interval when it is a decoration. Also comparisons.

Also an enquiry function "interval or decoration?" and conversions to/from parent type.

[*Note. ?? An alternative view on compressed intervals is to regard them as a flavor. When the threshold* τ *is* com, *they conform to the requirements of a flavor: they extend classical interval arithmetic, and one can tell when an arithmetic expression evaluation has failed to be common, because the result is a decoration instead of an interval. However, if* $\tau <$ com *this is no longer so.*]