8.8. The decoration system.

8.8.1. *Decorations and decorated intervals overview.* The decoration system of the set-based flavor conforms to the principles of Clause 6.1. An implementation makes the decoration system available by providing:

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

The system is specified here at a mathematical level, with the finite-precision aspects in §9.13. §8.8.2, 8.8.3, 8.8.4 give the basic concepts. §8.8.5, 8.8.6 define how intervals are given an initial decoration, and how decorations are bound to library interval operations to give correct propagation through expressions. §8.8.7 lists operations that do *not* propagate decorations. §8.8.8 discusses the decoration of user-defined arithmetic operations. §8.8.9 specifies the sixth decoration `com`, which is required in a multi-flavor implementation. §8.8.10 defines a restricted decorated interval arithmetic that suffices for some important applications and is easier to implement efficiently.

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

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:

$$d = \{\,(f,\boldsymbol{x}) \mid p_d(f,\boldsymbol{x}) \text{ is true}\,\}. \qquad (13)$$

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	$p_{\mathtt{dac}}(f,x)$	$\boldsymbol{x}$ 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);
`emp`	empty	$p_{\mathtt{emp}}(f,\boldsymbol{x})$	$\boldsymbol{x} \cap \mathrm{Dom}(f)$ is empty;
`ill`	ill-formed	$p_{\mathtt{ill}}(f,\boldsymbol{x})$	$\mathrm{Dom}(f)$ is empty.

(14)

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". The sixth decoration `com` (§8.8.9), if provided, lies at the top of the list as the "best" decoration of the set.

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 (13), `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. (15)

Namely the definitions (14) give:

`dac` $\subseteq$ `def` $\subseteq$ `trv` $\supseteq$ `emp` $\supseteq$ `ill`, note the change from $\subseteq$ to $\supseteq$; (16)

`dac` and `def` are disjoint from `emp` and `ill`. (17)

Hence for any $(f, \boldsymbol{x})$ there is a unique smallest (in the containment order (16)), 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:

$$\operatorname{dec}(f, \boldsymbol{x}) = d \iff p_d(f, \boldsymbol{x}) \text{ holds, but } p_e(f, \boldsymbol{x}) \text{ fails for all } e \subset d. \tag{18}$$

8.8.3. *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. An ill-formed decorated interval may also be 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.
[*Examples. The constructor call* `nums2interval`$(2, 1)$ *is invalid in this flavor, so its decorated version returns* NaI. *A compiler might determine that the constant expression* $[1, 1]/[0, 0]$ *comes from the undefined constant (zero-argument point function)* $1/0$, *so that it returns* NaI. *With more work, a compiler might determine that interval-evaluation of* $\sqrt{-1 - x^2}$ *should give* NaI, *for any input interval, because it is everywhere undefined as a real point function.*]

8.8.4. *Permitted combinations.* A decorated interval $\boldsymbol{y}_{dy}$ shall always be such that $\boldsymbol{y} \supseteq \operatorname{Rge}(f \mid \boldsymbol{x})$ and $p_{dy}(f, \boldsymbol{x})$ holds, for some $(f, \boldsymbol{x})$ as in §8.8.2—informally, it must tell the truth about some conceivable evaluation of a function over a box. If $dy =$ `dac` or `def` then by definition $\boldsymbol{x}$ is nonempty, and f is everywhere defined on it, so that $\operatorname{Rge}(f \mid \boldsymbol{x})$ is nonempty, implying $\boldsymbol{y}$ is nonempty. Hence the decorated intervals $\emptyset_{\texttt{dac}}$ and $\emptyset_{\texttt{def}}$ are contradictory: implementations shall not produce them.

No other combinations are essentially forbidden. However it is a consequence of a Level 2 requirement that the result $\boldsymbol{y}_{dy}$ of a constructor or a library arithmetic operation has $dy =$ `emp` or `ill` if and only if $\boldsymbol{y}$ is empty. Thus $\emptyset_{\texttt{trv}}$, and $\boldsymbol{y}_{\texttt{emp}}$ or $\boldsymbol{y}_{\texttt{emp}}$ with nonempty $\boldsymbol{y}$, are not generated by arithmetic expressions when initialized according to §8.8.5 and evaluated according to §8.8.6. However, they may be created by other means.

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

$$\texttt{newDec}(\boldsymbol{x}) = \boldsymbol{x}_d \qquad \text{where} \qquad d = \operatorname{dec}(\mathrm{Id}, \boldsymbol{x}) = \begin{cases} \texttt{dac} & \text{if } \boldsymbol{x} \text{ is nonempty,} \\ \texttt{emp} & \text{if } \boldsymbol{x} \text{ is empty,} \end{cases} \tag{19}$$

see §8.8.9 for the change to this if the `com` decoration is provided. For an already decorated interval $\boldsymbol{x}_{dx}$, `newDec()` discards the decoration and acts on the interval part,

$$\texttt{newDec}(\boldsymbol{x}_{dx}) = \texttt{newDec}(\boldsymbol{x}). \tag{20}$$

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

In this document a bare interval constant, in a context that expects a decorated interval, is implicitly *promoted* to a decorated interval by applying the `newDec` function. Whether an implementation provides such promotion at the program level is language-defined.
[*Example.* $\emptyset$ *is promoted to* `newDec`$(\emptyset) = \emptyset_{\texttt{emp}}$, *and* $[1, +\infty]$ *to* `newDec`$([1, +\infty]) = [1, +\infty]_{\texttt{dac}}$; *while* $[1, 2]$ *is promoted to* $[1, 2]_{\texttt{dac}}$ *if* `com` *is not supported, but* $[1, 2]_{\texttt{com}}$ *if it is.*]

8.8.6. *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 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

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

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**:

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

[*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*

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

   but also

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

   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 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, (29) 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], 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 (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*

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

   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 (27) is impractical to compute for some arguments $\boldsymbol{v}$. *Hence the weaker requirement (22) is made.*

]

8.8.7. *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 other versions of the operations that compute a bare interval result as above, and add a decoration suited to a particular application. How this is done is language- or implementation-defined.

In particular, to simplify defining functions piecewise, an implementation may define additional operations:

intersectionDec($\boldsymbol{x}_{dx}, \boldsymbol{y}_{dy}$) is as intersection($\boldsymbol{x}_{dx}, \boldsymbol{y}_{dy}$), except that it decorates the result with $\max(dx, dy)$.

convexHullDec($\boldsymbol{x}_{dx}, \boldsymbol{y}_{dy}$) is as convexHull($\boldsymbol{x}_{dx}, \boldsymbol{y}_{dy}$), except that it decorates the result with $\min(dx, dy)$.

A language may make either the standard operations, or these operations, its default operations for intersection and convexHull.

8.8.8. *User-supplied functions.* A user may define a decorated interval extension of some point function, as defined in §8.8.6, to be used within expressions as if it were a library operation.

[*Examples.*

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

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

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

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

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

Thus it is useful to code a tight interval extension by special methods, e.g. monotonicity arguments, and provide this as a new library function. Suppose this has been done. To convert it to a decorated interval extension just entails adding code to provide a local decoration and combine this with the input decoration by the min-rule (23). In this case it is straightforward to compute the strongest local decoration $d = \mathrm{dec}(f, \boldsymbol{x})$, *as follows.*

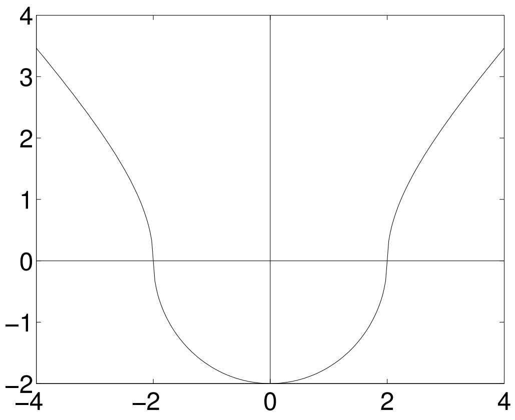
$$d = \begin{cases} \texttt{emp} & \text{if } \boldsymbol{x} = \emptyset \text{ or } \boldsymbol{x} = [0, 0], \\ \texttt{trv} & \text{if } 0 \in \boldsymbol{x} \neq [0, 0], \\ \texttt{dac} & \text{if } 0 \notin \boldsymbol{x} \neq \emptyset. \end{cases}$$

(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

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

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.

]

8.8.9. *The* `com` *decoration.* A multi-flavor implementation shall, and other implementations may, provide the sixth decoration `com` (see §6.3 in Chapter 1), 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.

(28)

Its position in the containment and propagation orders is given by

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

Including `com` causes the following changes to the decoration system:

– The strongest decoration of the identity function, $\mathrm{dec}(\mathrm{Id}, \boldsymbol{x})$ now equals `com` when $\boldsymbol{x}$ is bounded and nonempty.
– Hence the `newDec` function gives the decoration `com` instead of `dac` to a bounded, nonempty interval, namely

$$\texttt{newDec}(\boldsymbol{x}) = \boldsymbol{x}_d \quad \text{where} \quad d = \mathrm{dec}(\mathrm{Id}, \boldsymbol{x}) = \begin{cases} \texttt{com} & \text{if } \boldsymbol{x} \text{ is bounded and nonempty,} \\ \texttt{def} & \text{if } \boldsymbol{x} \text{ is unbounded,} \\ \texttt{emp} & \text{if } \boldsymbol{x} \text{ is empty.} \end{cases} \tag{29}$$

– Each arithmetic operation gives `com` as its local decoration if the conditions (28) are satisfied.

The propagation rule specified by (21, 22, 23) is unchanged. Note that minor changes are needed to the first example in §8.8.8.

[*Note. The* `com` *decoration describes both a Level 1 and a Level 2 property. When f is a library arithmetic operation φ, the computed interval $\varphi(\boldsymbol{x})$ means the enclosure of* $\mathsf{Rge}(\varphi\,|\,\boldsymbol{x})$ *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 $\mathsf{Rge}(f\,|\,\boldsymbol{x})$ *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 $\boldsymbol{x}$.*]

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

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 (14). 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 (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.

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` or `ill` is used instead.

At Level 2, different thresholds generate different compressed interval types. That is, if $\mathbb{T}$ is a parent type for compressed arithmetic, there shall be separate compressed interval types $\mathbb{T}_\tau$ for each threshold value $\tau \geq \texttt{trv}$. The only way to use compressed arithmetic with a particular threshold τ is to construct $\mathbb{T}_\tau$-intervals, that is, objects of type $\mathbb{T}_\tau$.

[*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" 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.*

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. "Mixed threshold" operations, combining compressed intervals of the same parent type and different threshold values, can be done in effect by first converting the input operands to the destination type, as described below. It is the user's responsibility to ensure that this is valid in the context of the application.*]

The enquiry function $\texttt{isInterval}(\boldsymbol{x})$ 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 $\tau\texttt{-compressedInterval}(\boldsymbol{X})$, where $\boldsymbol{X} = (\boldsymbol{x}, dx)$ is a decorated interval of the parent type, is a $\mathbb{T}_\tau$-interval as follows:

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

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

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

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

Conversion of a $\mathbb{T}_\sigma$-interval to $\mathbb{T}_\tau$-interval shall be equivalent to first converting to a normal decorated interval by `normalInterval()`, and then to the destination type by $\tau\texttt{-compressedInterval}(\boldsymbol{X})$. Such conversions need not be provided as single operations.

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 $\{\texttt{trv}, \texttt{def}, \texttt{dac}, \texttt{com}\}$ is treated (conceptually, not algorithmically) as an arbitrary $\boldsymbol{x}_d$ with nonempty interval $\boldsymbol{x}$ that is compatible with d: for d in $\{\texttt{trv}, \texttt{def}, \texttt{dac}\}$, $\boldsymbol{x}$ is unrestricted, while for $d = \texttt{com}$, $\boldsymbol{x}$ 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 (16)) 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 it 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.

[*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.*]