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).*

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 $\tau \geq$ `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` 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 $\tau \geq$ `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 `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 interval of the parent type, 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).
```

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 τ`-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 (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 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.*]