(*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 might 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—in general not a static object within program code, but derived dynamically from a particular program execution. 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 programming language.

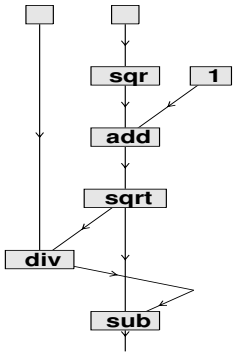
A *formal expression* defines a relation between certain mathematical variables—the *inputs*—and others—the *outputs*—via the application of named *operations*. It is by definition an acyclic (having an acyclic graph, see below) set of dependences between mathematical variables, defined by equations

$$v = \varphi(u_1, \ldots, u_k), \qquad (k \geq 0 \text{ being the } \textit{arity} \text{ of } \varphi) \tag{1}$$

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*; and distinct equations have distinct v's—the *single assignment* property.

Three 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



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. Form (c) has redundancy in the sense of repeated subexpressions.

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.)

(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, (c) is the form used in the FTIA proof in Clause C.8.

The three forms are computationally equivalent in the sense that *even in finite precision*, with identical inputs, each form computes identical (number or interval) intermediate values in some order—maybe with repetitions in form (c)—and identical output.

Here it is assumed that *evaluation* of any of these forms is, by definition, done using the operations exactly as written. E.g., one does not replace an occurrence of $x - x$ by 0, since that creates a different expression. In finite precision, it is assumed that at corresponding points of a graph, code list or algebraic expression the same version of the relevant operation is used. For example, storing and using intermediate results in IEEE 754 extended (80-bit) precision counts as using different operation-versions from using double (64-bit) precision throughout.

When evaluated in interval mode, multiple instances of the same variable can lead to excessive widening of the final result: e.g., evaluating $x - x$ with an interval input $\boldsymbol{x}$ gives, not $[0, 0]$, but an interval twice the width of $\boldsymbol{x}$. Thus the question, when it is valid to manipulate expressions—e.g., to replace $x - x$ by 0—is of especial importance for interval computation because of its potential to tighten enclosures. This is the *dependency issue*, covered in Annex **??**.

6.2. Function libraries. The formal operations are conceptually grouped into *libraries*, based on whether the variables denote scalars (the *point library*), bare intervals (the *bare interval library*), or decorated intervals (the *decorated interval library*), to reflect the three ways in which an expression can be evaluated. This conceptual view is independent of how the operations are presented in an actual computing environment: it could be via programming libraries, language primitives, infix operators, or other means, just as the standard is not concerned with the actual names or invocation methods. However, an implementation shall document how the formal operations are mapped to language entities.

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*.

In this standard, an implementation's library by definition comprises all its Level 2 versions of operations that it provides for any of its supported interval types. For the set-based flavor, these are specified in §10.6, 10.7, in §11.5, 11.6, 11.7 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 §11.8.

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* $-\infty < x \leq -1$ *and* $0 < x < +\infty$.]

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 that this 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 §11.8 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.