MOTION TO PROPOSE A "DISCONTINUOUS" DECORATION BIT VERSION 1

JOHN PRYCE

1. Introduction

1.1. **Terminology.** An *interval library function* means an interval version $\boldsymbol{f}(\boldsymbol{x}, \boldsymbol{y}, \ldots)$, provided by a P1788 implementation, of a mathematical real point function $f(x, y, \ldots)$. The set of these is the *interval library*. The conceptual set of mathematical f whose interval versions are provided will be called the *math library*.

Interval mappings that are not interval versions of point functions, such as "hull" and "intersection", are not interval library functions.

The math library need not all be—though usually will be—implemented in the *point library*, which is the set of floating-point functions provided by the implementation or its underlying floating-point system.

Mixed-format issues seem orthogonal to the present discussion, so I assume for now that all functions are implemented in the same interval datatype (format), e.g. inf-sup based on binary64.

1.2. **Background.** In May 2010 I circulated a position paper [5] intended to complement Nate Hayes' recent paper [4]. There, I proposed the following decoration bits:

1. Two "domain" bits (`domain`), for determining whether a function defined in terms of elementary functions is *everywhere defined*, or is *nowhere defined*, on its input box. (Nate Hayes is opposed to this scheme since he believes his tetrits do the same job better.)
2. A "discontinuous" bit (`discont`), for determining whether such a function is everywhere defined and continuous on its input box. (Nate supports this scheme.)
3. An "illformed" bit (`illform`), for determining whether an interval is nonsense because it is descended from the output of an invalid constructor call. (I believe Nate regards this as unnecessary. While not actively opposed to it, he argues that he carries in the two bits of a tetrit the information that I carry in three, the `domain` bits and the `illform` bit.)

This paper proposes a motion on the `discont` bit, upon which Nate and I agree.

2. Motion

The standard shall support the `discont` bit, as follows.

1. Decorated intervals shall include a sticky `discont` bit, which when correctly used evaluates the negative of the predicate $P(f, \boldsymbol{x})$ defined in Section 4 equation (1).
2. Interval library functions shall propagate `discont` as specified in Section 4, thus meeting requirement R2 in Subsection 3.1.
3. The P1788 group will design a generalised software architecture for language-level support of some or all of R1, R3, R4 in 3.1. The standard will *recommend* that any P1788-conforming language should support this architecture, either as an intrinsic language feature or as part of its interval library.

3. Rationale

3.1. **Difficulties.** Incorporating `domain` or `discont` in P1788 (and Hayes tetrits too IMO) poses problems we have not faced before, though I think similar ones will arise in future. The first difficulty is that these decorations are about deducing *dynamic, global* properties of code, namely of *a particular evaluation* of *a particular section of code*, `f`, by applying structural induction at run time. For the deduction to be valid these requirements must be met:

Date: August 21, 2010.

R1. The code f must represent an interval version $\boldsymbol{f}$ of a point function f built from math library functions e_i. Namely $\boldsymbol{f}$ is produced by implementing each e_i as its interval library version $\boldsymbol{e}_i$.
R2. The interval library functions must manipulate the decoration bits in the appropriate "sticky" way.
R3. The decoration bit(s) of each input argument (each component $\boldsymbol{x}_j$ of the input box $\boldsymbol{x}_1 \times \ldots \times \boldsymbol{x}_n$) must be suitably initialised on entry to f.
R4. The desired property value must be obtained by combining the relevant bits on each output arguments (component of output box) on exit from f.

R1 applies also to the Fundamental Theorem of Interval Arithmetic (FTIA), whose applicability in a program is also deduced by run time structural induction. However, the FTIA poses fewer problems of validity because R2–R4 do not apply to it.

The second difficulty is that P1788's primary remit is below the language level. It can specify the behaviour of individual operations, in particular can require R2, which is the responsibility of an implementor of the standard. But at this level it has *no control* over R1, R3 and R4. Here are some examples.

- A user may code an interval version of $f(x) = x^2 + 4x$ by writing the expression in two ways and taking the intersection:

```
interval f(interval x) {
  interval y1 = x*(x+4), y2 = (x+2)^2 - 4;
  return intersect(y1,y2);
}
```

  This gives a valid enclosure, but is not a function to which the theory of the `discont` bit applies: it violates R1.
- Depending on language and programming style, the inputs and outputs of an interval function $\boldsymbol{f}$ may appear in many guises. For instance $\boldsymbol{f}$ may be coded inline without explicitly being designated a function. Or its interval inputs may be separate arguments; or elements of a vector; or fields of an object; or any combination of these.
- Its outputs have similar possibilities, with the additional choice between output via one or (in a language such as Matlab) several return values, or output via the argument list, or a mixture of both.

In view of this I do not see how P1788 can provide features that *directly* support the enforcement of R1, R3, R4. To the extent such enforcement is possible, it can only be done at the language level. Here are some ways P1788 might approach this issue:

1. *Abdicate* any responsibility for correct use of these decoration features, either to the application programmer, or to a language standard if it chooses to take this on.
2. Design a generalised software architecture for supporting some or all of R1, R3, R4, and *recommend* that any P1788-conforming language should support it, either as an intrinsic language feature or as part of its interval library.
3. The same, but make it *required*.

Personally I think the first way would weaken the standard, seriously and for ever: we should not take it. The third is too rigid, hence too risky: we do not understand well how these features should best be supported even in common current languages and on current hardware. Whatever mechanism we design, someone will invent one that works better in some contexts.[1] Hence I support the second approach.

4. Summary of theory of sticky bits, from [5]

A. Assume a point function $y = f(x)$, where $x = (x_1, \ldots, x_n)$ and $y = (y_1, \ldots, y_m)$, defined by code f. For simplicity assume f has no loops or branches, though I think it is not hard to relax this. f can be described by its *computation graph* G, a directed acyclic graph (DAG) whose nodes are labeled $i = 1-n, \ldots, p+m$, each i being identified with a variable v_i. The n *input* nodes $v_{1-n}, \ldots, v_0$ are aliases for the x_j. Each non-input node represents a *statement* $v_i = e_i$(certain predecessors of v_i in G) for $i = 1, \ldots, (p + m)$, where e_i is a math library function. The last m variables $v_{p+1}, \ldots, v_{p+m}$ are

[1] Consider the history of concurrent process mechanisms: critical regions, semaphores, rendezvous, ...

aliases for the *outputs* $y_1, \ldots, y_m$. The other variables $v_1, \ldots, v_p$ are *intermediates*. This notation is used in Automatic Differentiation, see [2].

The v_i are also identified with the functions that they are of the inputs, e.g. an input v_{j-n} is the function $(x_1, \ldots, x_n) \mapsto x_j$ while an output v_{p+i} is the ith component $f_i(x_1, \ldots, x_n)$ of the final function f.

When `f` is evaluated over intervals or general sets, it has actual arguments $V_{j-n} = X_j$, $j = 1, \ldots, n$ that are sets, and each node of G is annotated with the set V_i that resulted from evaluating (the interval or set version of) e_i.

B. To be more precise, the x_j are the *nominated inputs* and the y_i are the *nominated outputs*. *Constants* are regarded as inputs: they are variables v_i whose function e_i has no arguments.

Considering real programming languages, P1788 needs to consider whether global variables (or Fortran `common`) are permitted as implicit inputs and/or outputs.

C. Those nodes that are on some path in G from an input to a nominated output are *live nodes*. The others are *dead nodes*, and their statements are *dead code*, which could be deleted without affecting the function.

D. The evaluation of `discont` is based on the following principle.

Structural induction. If for a property P:
– P is true at the input nodes of G;
– the truth of P at all predecessors of a live node implies its truth at that node;
then P is true at the nominated outputs.

E. **Continuity.** Interval algorithms that use fixed point theorems require to verify continuity, in the form "The restriction of f to box $\boldsymbol{x}$ is everywhere defined and continuous", that is the predicate

$$P(f, \boldsymbol{x}) = (\forall a \in \boldsymbol{x}) C(f, a, \boldsymbol{x}) \tag{1}$$

where

$C(f, a, \boldsymbol{x})$ = "The restriction of f to $\boldsymbol{x}$ is defined and continuous at a".

There are several detailed formalizations of $C(f, a, \boldsymbol{x})$; I propose the following.

$f(a)$ is defined, and for all $\epsilon > 0$ there exists $\delta > 0$ such that whenever x is in $\boldsymbol{x}$ and $f(x)$ is defined and $|x - a| \le \delta$, then $|f(x) - f(a)| \le \epsilon$.

[*Note. The "restriction" phrase is essential: consider the function $f(x) = floor(x) + \frac{1}{2}$. Its restriction to $\boldsymbol{x} = [1, 1.9]$ is everywhere defined and continuous, being the constant 1.5. Evaluating $\boldsymbol{y} = f(\boldsymbol{x})$ we find $\boldsymbol{y} \subseteq \boldsymbol{x}$ so we can deduce by Brouwer's Theorem that f has a fixed point in $\boldsymbol{x}$, which indeed it does. However the "unrestricted" f is not continuous at $1 \in \boldsymbol{x}$, since as $x \to 1$ from below, $f(x) = \frac{1}{2} \not\to f(1) = \frac{3}{2}$. Thus if we drop the "restriction" phrase we cannot deduce that the fixed point theorem applies in this example.*]

F. The value of $P(e_i, \boldsymbol{x}^{(i)})$ at the nodes of any dead code does not affect the result.

G. For reasons given in [5], `discont` propagates the negative of P. Specifically, its value on return from evaluating each e_k is "$P(e_i, \boldsymbol{x}^{(i)})$ is false at the current node or some predecessor of it" (i.e., for i equal to k or the index of some predecessor of node k). This is *initialised* to 0 at every (nominated or other) input node. It is *propagated* by evaluating $\neg P(e_i, \boldsymbol{x}^{(i)})$ and returning the "or" of this with the `discont` values of the inputs to e_i.

H. Elementary real analysis shows that if $P(e_i, \boldsymbol{x}^{(i)})$ holds at each live non-input node of the computation graph, then $P(f, \boldsymbol{x})$ holds.

Hence if, on exit from the function, `discont` is false at an output node, this proves the restriction to $\boldsymbol{x}$ of the corresponding component of f is everywhere defined and continuous.

This is only a *sufficient* condition: because of interval widening due to dependency, etc., it is possible to have continuity even though `discont` is true.

I. Most standard functions are either defined and continuous everywhere, or continuous precisely where they are defined; in either case they *always* set the raw value $\neg P(e_i, \boldsymbol{x}^{(i)})$ the same as the raw SuD "Somewhere unDefined" value of [5]. The few exceptions are less often used and logically simple: `sign, ceil, floor, nint, trunc` which are everywhere defined but have jump discontinuities.

As argued in [5], the value of SuD is a natural byproduct of the logic of an interval standard function. Hence, I expect the overhead of evaluating `discont` for all interval library functions to be negligible.

References

[1] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, *Efficiently computing static single assignment form and the control dependence graph*, ACM Transactions on Programming Languages and Systems, 13 (1991), pp. 451–490.

[2] A. Griewank, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, no. 19 in Frontiers in Appl. Math., SIAM, Philadelphia, Penn. (2000).

[3] N. Hayes, *Tetrits and "stickiness"*, email to P1788 (14 April 2010).

[4] N. Hayes, *Trits to Tetrits*, P1788 Position Paper (version of 28 May 2010).

[5] J.D. Pryce, *Decoration properties, structural induction, and stickiness*, P1788 Position Paper (version 3, 28 May 2010).