DECORATION PROPERTIES, STRUCTURAL INDUCTION, AND STICKINESS
VERSION 3

JOHN PRYCE

1. Introduction

This position paper is intended to complement Nate Hayes' recent paper *Trits to Tetrits* [4]. It is not quite a motion, but I make suggestions for discussion: see the Conclusions.

Decorations are optional attachments to intervals, chosen by P1788 as its exception-handling method. Loosely we think of exceptions as flagging "something unusual happened"; but looking more closely, we see their chief use in interval algorithms is to show a specific section of a run time execution trace, namely *a single function evaluation*, has specific properties—or not.

I use the term *prit* (property digit) to mean a field in a decoration that represents a property, and may be a bit, trit, tetrit, etc. A *sticky* prit is one that describes not just the current interval but some computation history. Definition 2 in [4] gives a rule for propagating tetrit values through interval operations, that is one definition of stickiness. All the prits that have been discussed—excluding the old "bounded" but including (I think) Dan Zuras' recent redefined "bounded", which is not discussed here—are sticky. I argue here that

1. A sticky prit enables code to deduce, at run time, a property of a function evaluation that is proved theoretically by *structural induction* over the function's computation graph. *(Fact)*
2. Discussing stickiness is futile unless the structural induction aspect is explicit. *(Opinion)*
3. Most prits we've considered are *pointwise*, meaning they relate to a predicate $P(f, x)$ where f is a function and $x = (x_1, \ldots, x_n)$ is quantified over an n-dimensional set $X = X_1 \times \ldots \times X_n$ of arguments to f. *(Fact)*
4. Nate's tetrit stickiness definition is not the only one: others might be useful. *(Opinion)*
5. Pointwise sticky bits are of two natural kinds, which I arbitrarily call "blue-sticky" and "red-sticky". My "domain" tetrit, the concatenation of a blue-sticky and a red-sticky bit, is locally equivalent to Nate's, but not globally equivalent since they propagate differently. *(Fact)*
6. Each of "blue-sticky" and "red-sticky" has four logically equivalent, all fairly natural, ways to express it mathematically in terms of quantifiers and the symbols $\wedge$ ("and") and $\vee$ ("or"). Each way suggests a style of writing code (probably at level 3, namely interval-library code) to implement stickiness. *(Fact)*
7. P1788 should mandate a particular style, to make code and specifications easier to understand. I make recommendations. *(Opinion)*
8. My "domain" prit describes the *outcome of a process*; it is not a property of an *individual interval*. In a popular analogy, it is about sequences of fence-panels, not fence-posts. *(Fact)*

   This leads me to conclusions about the right way to initialise it, that currently conflict with Nate's views, in particular with regard to the empty set.

I used to be in favour of global or regional flags to record "domain" and "discontinuous" properties. I give examples that have changed my view: they show a decoration-based implementation is robust in face of "dead code", in a way that global flags inherently cannot be.

I see no need for the normative standard text to mention material in this paper such as computation graphs, alternative options, etc.; but background theory on these lines should, IMO, go in an appendix to the standard.

Date: June 28, 2010.

Notation and terminology.

– General sets are written $X, Y, \ldots$; intervals in $\mathbb{R}$ or $\mathbb{R}^n$ are in bold italic $\boldsymbol{x}, \boldsymbol{y}, \ldots$.

– The symbols $\neg$, $\wedge$, $\vee$ denote logical "not", "and" and "or" respectively. I use 0 for "false" and 1 for "true".

– My usage of "everywhere" is the usual one in set theory and differs from Hayes'. Given a predicate $P(x)$, "P holds everywhere on X" means simply

$$(\forall x \in X)P(x),$$

where X may be empty (in which case the above is true for any P). Similarly "P holds somewhere on X" means $(\exists x \in X)P(x)$, and "P holds nowhere on X" is the negation of this, equivalent to "$(\neg P)$ holds everywhere on X".

– By contrast Hayes' meaning of "everywhere" is

$$(\forall x \in X)P(x) \wedge (\exists x \in X)P(x), \quad \text{equivalently} \quad (\forall x \in X)P(x) \wedge (X \neq \emptyset).$$

2. Structural induction

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` with no loops or branches. `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 an operation $v_i = e_i$(predecessors of v_i in G) for $i = 1, \ldots, (p+m)$, where e_i is an elementary function. The last m variables $v_{p+1}, \ldots, v_{p+m}$ are aliases for the *outputs* $y_1, \ldots, y_m$. The other variables $v_1, \ldots, v_p$ are *intermediates*.

This description of code, and the v_i notation, is standard in Automatic Differentiation, see [2]. It is equivalent to *static single-assignment* (SSA) form, see [1].

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.

As said above, sticky prits are a way to deduce, at run time, properties that are proved by applying to G the principle of

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

Though true, this is complicated by the fact that the input and output nodes of G may not comprise just the *nominated* ones $x_1, \ldots, x_n$ and $y_1, \ldots, y_m$. Figure 1 shows the silly but valid code of a function f, written in SSA form, with its graph alongside.

This function is to be evaluated in interval mode, with each point operation replaced by its decorated-interval version. It illustrates some issues that decoration processing must deal with:

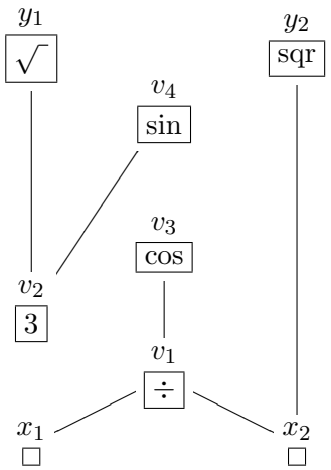
(a) Node v_2 is not a nominated input, but is an input node of the DAG nonetheless. Hence, its decoration must be initialised—how?
(b) Node v_3 is not a nominated output, but is an output node of the DAG nonetheless. A zero-divide exception might happen on the way to v_3—does this matter to f?
(c) Input x_1 does not affect the outputs, nor is output y_1 affected by the inputs. Intermediates v_1, v_3, v_4 do not affect the outputs.

A compiler, by static code analysis, could mark the lines that set v_1, v_3 and v_4 as "dead code" and delete them; but our exception handling cannot assume that.

My answer to the question in item (b) is No. For the prits that concern us here—"domain", "discontinuous" and possibly the Zuras "bounded"—what matters is the process of computing f values. So the only exceptional events to record are those *on some path from inputs to nominated*

function $[y_1, y_2] = f(x_1, x_2)$
$v_1 = x_1/x_2$;
$v_2 = 3$;
$v_3 = \cos(v_1)$;
$v_4 = \sin(v_2)$;
$y_1 = \sqrt{v_2}$;
$y_2 = x_2^2$;
end

(pseudo-Matlab notation.)



Graph conventions.

The nominated input nodes, at bottom level, show just the names x_1, x_2 of the inputs.
Each other node shows the elementary function performed and the named result.
A literal, like 3, is regarded as naming an elementary function with no arguments, that returns the indicated value.
The nominated outputs y_1, y_2 are put at the top level.

FIGURE 1. Silly but valid function

outputs, where "inputs" must include both nominated inputs and constants like 3 in the above function. Note the asymmetry between input and output this introduces.

I regard the above as a design decision, and not a total no-brainer—an opposing view is reasonable in some contexts.

Thus P1788 exception handling is based on the following amended induction principle. Define a node to be *live* if it lies on a path to the nominated outputs (otherwise *dead*).

Structural induction, revised. 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.

Recording exceptions this way is what an implementation based on propagating sticky prits from inputs to outputs *automatically* does, and is what a global-flag implementation *cannot* do.

3. Blue and Red stickiness

This paper is mainly concerned with *pointwise* properties. Such a property is associated with a predicate $P(\varphi, \xi)$ where φ is some function and ξ represents its arguments ξ_i, as a tuple or a comma-separated list whichever is convenient.

[*Example.* Let P be "φ is defined at ξ", and φ be real division $\div(\xi_1, \xi_2) = \xi_1/\xi_2$. Then "the defined-ness of 2/0" can be written as $P(\div, (2, 0))$ or as $P(\div, 2, 0)$, which evaluates to 0.]

[*Example.* "illformed" is a property of intervals, but not a pointwise one. The other properties discussed here are pointwise, though "discontinuous" needs some qualifications—see later.]

3.1. **Hayes' example.** Nate Hayes illustrates the needs of his branch-and-bound algorithm by a, not everywhere defined, scalar function $y = f(x)$ where $x = (x_1, \ldots, x_n)$. (In his example $n = 2$, but this does not matter.) Given a box (product of intervals) $\boldsymbol{x}$ in $\mathbb{R}^n$, one does

– if f is found to be everywhere defined on $\boldsymbol{x}$, and to satisfy some other property such as being ≤ 0 on $\boldsymbol{x}$, then accept $\boldsymbol{x}$;
– if f is found to be nowhere defined on $\boldsymbol{x}$ then discard $\boldsymbol{x}$;
– if the diameter of $\boldsymbol{x}$ is below a threshold then mark $\boldsymbol{x}$ indeterminate;
– else bisect $\boldsymbol{x}$ and do the same thing to the pieces.

That this is practical to implement, relies on two structural-induction facts applied to the computation graph G of f when it is evaluated on an input box $\boldsymbol{x} = \boldsymbol{x}_1 \times \ldots \times \boldsymbol{x}_n$. Expressed in ordinary language they are as follows, where an elementary operation e being "live" means its node is live in the DAG sense defined above.

StrInd1. (**every** live operation e in G is **everywhere defined** on its input box) $\Rightarrow$ f is **everywhere defined** on $\boldsymbol{x}$;
or in an equivalent form that is more convenient later:
(**every** live operation e in G is **nowhere undefined** on its input box) $\Rightarrow$ f is **nowhere undefined** on $\boldsymbol{x}$.

StrInd2. (**some** live operation e in G is **nowhere defined** on its input box) $\Rightarrow$ f is **nowhere defined** on $\boldsymbol{x}$.

In absence of better names I call these "blue-sticky" and "red-sticky" induction, respectively.

[*Note. In both cases, the reverse implication is false in general, but the idea behind the bisection is that the smaller $\boldsymbol{x}$ is, the more often the reverse implications* do *hold. For vector-valued f it is agreed that $y = f(x)$ is undefined if any of its components is undefined.*]

3.2. **Formalisation.** Let $P(\varphi, \xi)$ be a predicate as above. We call its negation Q:

$$Q(\varphi, \xi) = \neg P(\varphi, \xi),$$

and do not *(prima facie)* favour one over the other. Hence we do not favour existential ($\exists$) over universal ($\forall$) quantifiers—for instance $(\exists\xi)Q(\varphi, \xi)$ is logically the same as $\neg(\forall\xi)P(\varphi, \xi)$—though Hayes [4, 2.1] standardises on the former.

With the notation and definitions of Section 2, let $P(\varphi, \xi)$ mean "φ is defined at ξ", so $Q(\varphi, \xi)$ means "φ is undefined at ξ". Let box $\boldsymbol{x}$, the product of intervals $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$, be an actual input to the code f. For each node $i = 1, \ldots, p + m$ of G let $\boldsymbol{x}^{(i)}$ be the resulting actual input to operation e_i. Thus $\boldsymbol{x}^{(i)}$ is the product of n_i intervals $\boldsymbol{v}_k$, where k runs over the n_i predecessors of i in G, with n_i being the arity (number of arguments) of e_i. Then "e_i is everywhere defined on its input box" may be written

$$\big(\forall x \in \boldsymbol{x}^{(i)}\big) P(e_i, x);$$

hence the left side, p^*, of the implication in StrInd1 above may be written

$$p^* = \Big(\big(\forall x \in \boldsymbol{x}^{(1)}\big) P(e_1, x) \ \wedge\ \big(\forall x \in \boldsymbol{x}^{(2)}\big) P(e_2, x) \ \wedge\ \ldots \ \wedge\ \big(\forall x \in \boldsymbol{x}^{(p+m)}\big) P(e_{p+m}, x)\Big). \qquad (1)$$

The truth of p^* implies the conclusion

$$(\forall x \in \boldsymbol{x}) P(f, x), \qquad \text{"}f \text{ is everywhere defined on } \boldsymbol{x}\text{"}.$$

Similarly, by definition $Q(\varphi, \xi)$ means "φ is undefined at ξ", so "e_i is nowhere defined on its input box" may be written

$$\big(\forall x \in \boldsymbol{x}^{(i)}\big) Q(e_i, x);$$

hence the left side, q^*, of StrInd2 may be written

$$q^* = \Big(\big(\forall x \in \boldsymbol{x}^{(1)}\big) Q(e_1, x) \ \vee\ \big(\forall x \in \boldsymbol{x}^{(2)}\big) Q(e_2, x) \ \vee\ \ldots \ \vee\ \big(\forall x \in \boldsymbol{x}^{(p+m)}\big) Q(e_{p+m}, x)\Big), \qquad (2)$$

The truth of q^* implies the conclusion

$$(\forall x \in \boldsymbol{x}) Q(f, x), \qquad \text{"}f \text{ is nowhere defined on } \boldsymbol{x}\text{"}.$$

One could shorten the $\wedge \ldots \wedge$ in (1), and the $\vee \ldots \vee$ in (2), by quantifying over $i = 1, \ldots, p+m$; but I do not do this, because it is of a different nature from the quantifications over each $\boldsymbol{x}^{(i)}$. Each of the latter is a real-analysis assertion about the behaviour of an elementary function e on a certain set of inputs; it is implemented by correct coding of a *library function*—the interval version $\boldsymbol{e}$ of e. The former is evaluated by doing structural induction at run time, while evaluating the *user function* f on the actual inputs $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$.

4. Rewritings of StrInd1 and StrInd2

4.1. **A schematic form.** To show the structure, write the right side of (1) schematically as

$$(\forall)P \wedge (\forall)P \wedge \ldots \wedge (\forall)P.$$

Using $Q = \neg P$ and the rules for negation we can write this in several equivalent ways, which schematically are:

Versions of blue-sticky induction

$$p^* = \Big((\forall)P \wedge (\forall)P \wedge \ldots \wedge (\forall)P\Big), \tag{3}$$

$$\neg p^* = \Big(\neg(\forall)P \vee \neg(\forall)P \vee \ldots \vee \neg(\forall)P\Big), \tag{4}$$

$$p^* = \Big(\neg(\exists)Q \wedge \neg(\exists)Q \wedge \ldots \wedge \neg(\exists)Q\Big), \tag{5}$$

$$\neg p^* = \Big((\exists)Q \vee (\exists)Q \vee \ldots \vee (\exists)Q\Big). \tag{6}$$

Doing the same for (2) gives:

Versions of red-sticky induction

$$q^* = \Big((\forall)Q \vee (\forall)Q \vee \ldots \vee (\forall)Q\Big), \tag{7}$$

$$\neg q^* = \Big(\neg(\forall)Q \wedge \neg(\forall)Q \wedge \ldots \wedge \neg(\forall)Q\Big), \tag{8}$$

$$q^* = \Big(\neg(\exists)P \vee \neg(\exists)P \vee \ldots \vee \neg(\exists)P\Big), \tag{9}$$

$$\neg q^* = \Big((\exists)P \wedge (\exists)P \wedge \ldots \wedge (\exists)P\Big). \tag{10}$$

There is no way to reduce one kind to the other, by replacing the underlying predicate with its negation or by other manipulations.

Hayes prefers existential quantifiers, so he would choose one of (5, 6, 9, 10). For instance (6), in ordinary language, gives the negation of p^* as "some operation in G is somewhere undefined", which is clearly a valid restatement.

Each version suggests a style of coding the structural induction by sticky decoration bits: the styles differ at the "bit-twiddling" level but are equivalent. The bits associated with the two kinds of induction will be called "blue bits" and "red bits" respectively.

For example, to code (3), decorate intervals with a blue bit p that is initialised to 1 on all the inputs. Each e_i computes $\big(\forall x \in \boldsymbol{x}^{(i)}\big)P(e_i, x)$ and sets the blue bit on its output to the "and" of this with the blue bits of its inputs. At the end, the "and" of all the blue bits of the outputs $\boldsymbol{y}_1, \ldots, \boldsymbol{y}_m$ gives p^*.

To code (6), set the blue bit to 0 on the inputs. Each e_i computes $\big(\exists x \in \boldsymbol{x}^{(i)}\big)Q(e_i, x)$—the negation of what (3) does—and sets the blue bit on its output to the "or" of this with the blue bits of its inputs. At the end, the "or" of all the blue bits of the outputs $\boldsymbol{y}_1, \ldots, \boldsymbol{y}_m$ gives $\neg p^*$.

4.2. **Discussion in relation to the standard.**

a. The $(\forall)P$'s, $(\exists)P$'s, etc., and the way sticky bits propagate through operations, are coded by people who write interval libraries, so their code is not visible at user program level.
b. The initialising of inputs, and final "and"ing or "or"ing, happens at user program level. A language or an interval package could—IMO should—offer utility functions to support this.
c. For structural induction to work on a user function, library functions must all use one style.
d. In most computing, initialising a value to 0 is more natural than initialising it to 1. The options that initialise thus are those with $\vee \ldots \vee$. Of these the simplest are (6) and (7).
e. Therefore I propose P1788 require blue-sticky bits to be implemented in style (6), and red-sticky bits in style (7). This implies sticky bits on inputs are always initialised to 0.
f. Hence the value returned to the user shall be $\neg p^*$ for blue-sticky bits, q^* for red-sticky bits. In case of an interval function with vector output, the result is spread over the output components; the final result comes from "or"ing the relevant bits, whether they are blue or red.

5. Some examples

We follow our version of the "domain" tetrit through some function evaluations. We define it as the pair of bits (ND, SuD), written as a 2-bit number such as 01. Here ND means "Nowhere Defined" and SuD means "Somewhere unDefined" on some set X, i.e. $\neg(\exists x \in X)D(\phi, x)$ and $(\exists x \in X)\neg D(\phi, x)$ respectively where $D(\phi, x)$ means "function ϕ is defined at x". This follows the form recommended above, since the "normal" or "success" case yields (ND, SuD) = 00, meaning f is "Somewhere Defined, Nowhere unDefined", i.e. defined everywhere on its nonempty input set, (= "everywhere" in the Hayes sense).

Notation. We write, e.g., $\boldsymbol{x} = [1, 2]_{\texttt{D01}}$ (D for "domain" to identify the prit) to denote a decorated interval whose interval part is $[1, 2]$ and that has (ND, SuD) = 01.

5.1. **The silly function of Figure 1.** We evaluate (the interval version of) f at $\boldsymbol{x} = ([1, 2], [0, 0])$, assuming exact arithmetic. The Sticky column shows the result prit computed as the "or" of the current operation prit and those of the inputs, with the latter in slanted text.

Var	Op		Value	Sticky	Note
$\boldsymbol{x}_1$ =		=	$[1, 2]_{\texttt{DOO}}$		Standard initialisation.
$\boldsymbol{x}_2$ =		=	$[0, 0]_{\texttt{DOO}}$		" "
$\boldsymbol{v}_1$ =	$\boldsymbol{x}_1/\boldsymbol{x}_2$	=	$\emptyset_{\texttt{D11}}$	11∨*00*∨*00*	÷ is nowhere defined, somewhere undefined on $[1, 2] \times [0, 0]$.
$\boldsymbol{v}_2$ =	3	=	$[3, 3]_{\texttt{DOO}}$		Standard initialisation.
$\boldsymbol{v}_3$ =	$\cos(\boldsymbol{v}_1)$	=	$\emptyset_{\texttt{D11}}$	10∨*11*	cos() is nowhere defined, nowhere undefined on $\emptyset$.
$\boldsymbol{v}_4$ =	$\sin(\boldsymbol{v}_2)$	=	$[\sin(3), \sin(3)]_{\texttt{DOO}}$	00∨*00*	sin() is somewhere defined, nowhere undefined on $\boldsymbol{v}_2$.
$\boldsymbol{y}_1$ =	$\sqrt{\boldsymbol{v}_2}$	=	$[\sqrt{3}, \sqrt{3}]_{\texttt{DOO}}$	00∨*00*	Similar.
$\boldsymbol{y}_2$ =	$\boldsymbol{x}_2^2$	=	$[0, 0]_{\texttt{DOO}}$	00∨*00*	Similar.

On "or"ing the tetrits of the two outputs we obtain, at least conceptually, the decorated vector

$$\boldsymbol{y} = ([\sqrt{3}, \sqrt{3}], [0, 0])_{\texttt{DOO}},$$

showing "success". Some points worth noting:

(a) As desired, the "bad decorations" on $\boldsymbol{v}_1$ and $\boldsymbol{v}_3$ do not show up in the output; this would not be the case were global flags used.

(b) In our scheme the two bits ND, SuD are propagated independently of each other (not the case for the Hayes bits (P^+, P^-)). Each independently gives a fact at the end—or not:
- ND = 1 is a result: f is nowhere defined on the input box. ND = 0 gives no result.
- SuD = 0 is a result: f is everywhere defined on the input box. SuD = 1 gives no result.

(c) More than one value of the domain tetrit is possible for the empty set. In this example, $\boldsymbol{v}_1$ and $\boldsymbol{v}_3$ are both empty, with decorations D11 and D10 respectively.

Indeed, evaluating f at $\boldsymbol{x} = (\emptyset, [0, 0])$ shows a third possibility, namely $\boldsymbol{x}_1 = \emptyset_{\texttt{DOO}}$ after standard initialisation. The output is as before, namely $\boldsymbol{y} = ([\sqrt{3}, \sqrt{3}], [0, 0])_{\texttt{DOO}}$. When we recall our prit describes a process, not the result interval, this is unsurprising.

(d) What about the interval version of a constant, if it had been a value like 0.3, not exactly representable in a binary format? I regard this as a language issue and skate past it.

5.2. **Relation to Hayes' domain tetrit.** This section has been revised following discussions with Hayes. I apologise for any remaining misconceptions of his scheme.

Of course, our pair ND, SuD is equivalent to Hayes' P^+, P^- in [4] on the *individual operation* level, at which ND = $\neg P^+$ and SuD = P^-. Before giving further examples we show, in the table below, the relation between them. It also expresses Hayes' 3-2-1-0 priority rank in terms of the tetrit, essentially as in Hayes [3]. We use boolean algebra notation: $'$ is logical not, and

+ is addition modulo 2, alias `xor`.

Status		(ND, SuD)	(P^+, P^-)	(ND′, ND′+SuD)	=	Priority
Somewhere defined	Nowhere undefined	00	10	11	=	3
Somewhere defined	Somewhere undefined	01	11	10	=	2
Nowhere defined	Somewhere undefined	11	01	01	=	1
Nowhere defined	Nowhere undefined	10	00	00	=	0

However, the Hayes tetrit propagates differently from ours in an *extended computation* because its "stickiness rule" is to take the least[1] (normal integer sense) of the priority returned by the current operation and those of the inputs, by contrast to the bitwise "or" of ours. Here are the op-tables for propagation of both methods, expressed in terms of priority:

Hayes (P^+, P^-)

inf	3	2	1	0
3	3	2	1	0
2	2	2	1	0
1	1	1	1	0
0	0	0	0	0

Pryce (ND,SuD)

	00	01	11	10
"or"	3	2	1	0
00, 3	3	2	1	0
01, 2	2	2	1	1
11, 1	1	1	1	1
10, 0	0	1	1	0

So one might say the difference is technical, in how priority 0 interacts with priorities 1 and 2. However, the more important difference is in meaning.

The (ND, SuD) tetrit merely records whether, in a specific segment of computation, certain events occurred. It is clear from [4] (e.g. 2.2) that the Hayes tetrit has a similar aim. But also [Hayes, private email 2010/06/25]: (a) he has deliberately left the description of initialisation to a separate paper, so it is currently not clearly specified what segment of computation is monitored; (b) priority 0 is used to denote "illformed" (formerly "invalid"). I personally view "illformed" as an attribute of the interval-in-itself rather than of the history that produced it; and the Hayes scheme yields results I find strange, as illustrated by the example

Compute $f(\boldsymbol{x})$ where $f(x) = \exp(\log(x))$ and $\boldsymbol{x} = [-2, -1]$.

The following table shows the decorations produced by the two schemes, on evaluating this.

Var	Op	Pryce	Hayes	Note
$\boldsymbol{x}$	=	$= [-2, -1]_{\texttt{D00=P3}}$	$[-2, -1]_{\texttt{P3}}$	Assuming Hayes initialises as we do.
$\boldsymbol{v}$	$= \log(\boldsymbol{x}) =$	$\emptyset_{\texttt{D11=P1}}$	$\emptyset_{\texttt{P1}}$	log() is nowhere defined, somewhere undefined on $\boldsymbol{x}$.
$\boldsymbol{y}$	$= \exp(\boldsymbol{v}) =$	$\emptyset_{\texttt{D11=P1}}$	$\emptyset_{\texttt{P0}}$	exp() is nowhere defined, nowhere undefined on $\boldsymbol{v}$.

The final result $\emptyset_{\texttt{P1}}$ in our scheme just says "at some stage, a function was nowhere defined, somewhere undefined on its input". On the Hayes scheme, the empty, priority-1, value of $\boldsymbol{v}$ has converted to an empty, priority-0, value of $\boldsymbol{y}$. That is, the final $\boldsymbol{y}$ is recorded as illformed. More generally, applying *any* algebraic function (that is, interval extension of a point-function) to an empty input has this result because the current operation necessarily gives $(P^+, P^-) = (0, 0)$, i.e. priority 0. The "inf" propagation rule thus gives priority 0 on the output.

Hayes has explained why he does not regard this result as contradictory, but I find it far from intuitive. My feeling is that by combining history data with properties of the interval-in-itself, the Hayes scheme is trying to pack more than two bits of information into two bits.

Until initialisation is specified for the Hayes scheme, I am also not sure how it handles the "plug $\boldsymbol{y}$ back into g" aspect of the next example.

[1] Hayes shows this is equivalent to the & operation on the `bool_set` type, but we do not need this here.

5.3. **A more normal function.** Let $g(x) = 1 + \sqrt{x}$, and evaluate $\boldsymbol{y} = g(\boldsymbol{x})$ where $\boldsymbol{x} = [-1, 4]$. Converting g to standard form we have

Var	Op	Value	Sticky	Note
$\boldsymbol{x}_1 =$		$= [-1, 4]_{\texttt{DOO}}$		Standard initialisation.
$\boldsymbol{v}_1 =$	$\sqrt{\boldsymbol{x}_1}$	$= [0, 2]_{\texttt{D01}}$	$01 \vee 00$	$\sqrt{}$ is somewhere defined, somewhere undefined on $[-1, 4]$.
$\boldsymbol{v}_2 =$		$= [1, 1]_{\texttt{DOO}}$		Standard initialisation.
$\boldsymbol{y}_1 = \boldsymbol{v}_1 + \boldsymbol{v}_2 =$		$[1, 3]_{\texttt{D01}}$	$00 \vee 01 \vee 00$	

giving $\boldsymbol{y} = [1, 3]_{\texttt{D01}}$. We may be seeking a fix-point of g, which exists at $x = (3+\sqrt{5})/2 \approx 2.618$. To this end we may wish to plug $\boldsymbol{y}$ back into g and find $\boldsymbol{z} = g(\boldsymbol{y})$. At this point the existing decoration D01 on $\boldsymbol{y}$ *is irrelevant*—it must be re-initialised to D00 to continue. Then we do:

Var	Op	Value
$\boldsymbol{x}_1 =$		$= [1, 3]_{\texttt{DOO}}$
$\boldsymbol{v}_1 =$	$\sqrt{\boldsymbol{x}_1}$	$= [1, \sqrt{3}]_{\texttt{DOO}}$
$\boldsymbol{v}_2 =$		$= [1, 1]_{\texttt{DOO}}$
$\boldsymbol{y}_1 = \boldsymbol{v}_1 + \boldsymbol{v}_2$		$= [2, 1 + \sqrt{3}]_{\texttt{DOO}}$

giving $\boldsymbol{z} = [2, 1+\sqrt{3}]_{\texttt{DOO}} \approx [2, 2.732]$. The SuD bit in this case is identical to the "discontinuous" bit SuC below, and indicates that g is everywhere defined and continuous on $\boldsymbol{y}$. Since also $\boldsymbol{z} \subseteq \boldsymbol{y}$, we deduce that there is a fix-point of g within $\boldsymbol{z}$, as is indeed the case.

5.4. **Implications of triviality.** "Edge cases" are instructive. Consider the trivial function $h(x) = x$, and evaluate $\boldsymbol{y} = h(\boldsymbol{x})$ where $\boldsymbol{x}$ is the empty set.

The standard code form does not allow the input x_1, aka v_0, to be the same variable as the output y_1, aka v_1. Thus, apart from initialising x_1, there must be one line of code: $y_1 = x_1$. Execution looks like this:

Var	Op	Value	Note
$\boldsymbol{x}_1 =$		$= \emptyset_{\texttt{DOO}}$	Standard initialisation.
$\boldsymbol{y}_1 = \boldsymbol{x}_1$		$= \emptyset_{\texttt{D10}}$	D10 because h is nowhere defined, nowhere undefined on $\emptyset$.

giving $\boldsymbol{y} = \emptyset_{\texttt{D10}}$. We deduce *simple assignment can change the decoration.* This is natural, as assignment consists of applying the identity function $i(x) = x$ (same as h). This must be regarded as an elementary function, and given an interval version following normal rules. In particular $i()$ is nowhere defined, nowhere undefined on $\emptyset$, and should set the decoration accordingly.

6. 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".

The "restriction" phrase is essential: consider the function $f(x) = \text{floor}(x) + \frac{1}{2}$. 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.

For these applications we just need one decoration bit. It is a little more complicated than "defined" because of the need to mention the input box $\boldsymbol{x}$. The underlying predicate is

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

This can be formalised in more than one way; Dan Zuras, with good reason, disliked one way I suggested. However all the ways I have tried *are exactly equivalent as far as intervals are concerned* (they differ on more complicated sets). The following is one way.

> $f(a)$ 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 $|f(x) - f(a)| \le \epsilon$.

With the notation used above, it is easy to prove that if $(\forall a \in \boldsymbol{x}^{(i)})C(e_i, a, \boldsymbol{x}^{(i)})$ holds at each live non-input node of the computation graph, then we conclude that $(\forall a \in \boldsymbol{x})C(f, a, \boldsymbol{x})$: the restriction of f to $\boldsymbol{x}$ is everywhere continuous.

Thus, applications that require to prove continuity can do it with one (blue-sticky) bit that I call "discontinuous" and denote by SuC, "Somewhere unContinuous", initialised to 0 as usual.

Many standard functions are either defined and continuous everywhere, or continuous precisely where they are defined; in either case they *always* set SuC to the same value as SuD. The few exceptions are less often used and logically simple: `sign, ceil, floor, nint, trunc` which are everywhere defined but have jump discontinuities. Thus, I believe that SuC can be implemented on top of SuD with almost no extra code and insignificant run time overhead.

7. Well-formedness

Not all important prits describe process. The "wellformed/illformed" (formerly "valid/invalid") prit does not, even though it can only be defined by induction.

Namely, now we have the decoration mechanism, I see no sense in letting an invalid constructor call such as "the interval from NaN to 2" return any normal interval. It should return an object that is clearly marked as "nonsense". Recursively, the result of any interval operation with a nonsense input should also be so marked.

A parallel or vector computation may construct large arrays of such intervals (from, say, pairs of reals) with some nonsense ones scattered among them. In that situation, marking them as nonsense provides a totally foolproof way to recognise them, which one cannot achieve from a scheme based on setting the object to `Empty` or `Entire` or whatever.

Such an "illformed" mark is essentially equivalent to Not an Interval, NaI, but can be implemented *very* cheaply using decorations, far more so than can an NaI value encoded in the interval part of an object—as pointed out in the Hayes–Neumaier motion 8 paper.

The Hayes scheme aims to encode illformed-ness as one of the values of the domain tetrit. It is easy to see that this cannot be done in my scheme; it should be a separate "illformed" bit—sticky, but not pointwise and therefore neither "blue" nor "red"—within a decoration.

Much of the behaviour of illformed objects is for us to choose, rather than dictated by the mathematics. For example, how do they behave in comparisons? What is returned when they are arguments to point-valued functions such as midpoint? My view: an illformed interval should be like a "black hole" with no usable properties; but these issues need a separate motion.

8. Conclusion

My specific conclusions and suggestions are:

A. The P1788 group should not, at this stage, require all decoration prits to be of the same kind: bits, trits, tetrits, etc. should all be considered on their merits.
B. Sticky bits should be implemented in such a way that they are initialised to zero.
C. The "domain" prit, giving sufficient conditions for a function to be everywhere, or nowhere, defined, can be implemented as two such bits ND, SuD.
D. The "discontinuous" prit, giving sufficient conditions for a function to be everywhere defined and continuous, can be implemented as another such bit SuC, which most library functions *always* set the same as SuD.
E. There should be an "illformed" sticky bit.

So I propose 4 bits of decoration, with possibly more to be added.

About naming. When teaching these concepts it is natural to discuss "wellformed-ness" and "continuity"; but if one adopts the principle of initialising to 0 (false) and propagating by "or", the actual properties being recorded are the opposite of these. So pedagogically, "illformed" and "discontinuous" seem better names. The name "domain" is neutral and seems appropriate.

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).