A Type Theory for the Documentation of PureData

Mathieu Bouchard

matju@artengine.ca

ABSTRACT

Types are classifications of values (and of holders of values) that exist for various reasons: as a mapping to lower-level data, as a way to automatically catch errors, as a way to express constraints, requirements, expectations. In this paper, all of these are considered, and furthermore, it is considered that typing is a notion that is fairly flexible, that it is a concept that can be used well beyond its usual use, especially compared to how it is used in PureData.

To get to such a definition, this paper deconstructs the notion of type as it is commonly understood. This leads to the understanding that types are expectations that we use to make sense of what we are creating. Types can make the documentation more efficient by becoming a specialised vocabulary that is used in the documentation. The preexisting use of the word *type* in the context of PureData is something completely different that has to do with constraining rather than explaining. This paper describes workarounds for type-checking-induced limitations, using parametrised class names.

The existing documentation of PureData, including documentation of all (or almost all) external libraries, and including PDDP[1], do not use such a concept of type: in that methodology, every class of *t_ object* is given a help file, and anything else has to be covered by special tutorials; several topics are not given explicit names, instead being paraphrased in documentation, which is a cause of redundancy (including non-obvious redundancy).

This paper proposes remedies to these some of those documentation problems and uses those solutions as a foundation toolbox which also connects into formal verification and other forms of auto-testing. With supporting methodology, this in turn improves the process of improving PureData at a technical level. (This paper doesn't claim to offer anything about problems that may occur at the social level)

Keywords

add your keywords here

Pd Convention 2007. Montréal, Québec, Canada.

1. TYPOLOGY

1.1 Conceptions of the Meaning of Types

Over the years, various notions of types have been used in programming languages. Here's a brief evolution of the notion, not supposing that those steps are all ordered according to improvement, nor supposing that it is always possible to strictly order them:

Machine Code and Assembly Language: a type is the size of a chunk of data, counted in bits.

Fortran and Basic: a type is what distinguishes integer addition from floating-point addition. (built-in polymorphism that cannot be extended by the user) Also distinguishes numbers from strings. There are operations that work only on numbers and others that work only on strings. Arrays are not values and can't necessarily even be considered as variables.

C: a type serves all of the above purposes, and to distinguish signed vs unsigned, integer vs pointer, structs vs simple values, and pointers among themselves according to what they point to. There are parametrised types, but they are limited to two templates (pointer types and array types) that are built into the syntax and not extensible.

Lisp: a type is something that belongs to the value contained in the variable, not to the variable itself. Therefore each value carries a tag identifying its own type, instead of requiring that information to be decided in advance for each variable.

Simula67: a struct type does not only define a shape of structs (in terms of a list of different variables) but also defines a set of procedures (methods) that act in the context of structs of that type. A struct type may have subtypes (that have more variables, more methods or different methods) and structs of that subtype are considered to be also part of the type that it is a subtype of (the supertype). This was the beginning of object-oriented programming 40 years ago (and still looks surprisingly familiar).

Smalltalk: much like Simula67 but more like Lisp (variables are never typed at all) and without any distinction between base types vs struct types: variables contains pointers to objects, or perhaps nonpointers, but users don't even have to know the difference. This is the beginning of *pure* object-oriented programming. Ruby and Python 2.2 are just the same.

Ada and C++: much like Simula, but one can define custom parametrised types.

ML/Haskell: very much relying on custom parametrised types; also, most characteristically, type is not checked, it's

inferred, so that you don't even have to write the types, even though every computation has a type at compile time.

TCL: At the level of using this language, types don't exist: everything is a string. The main implementation uses an internal format made of a string and/or some other structure that is easier/faster to handle from the C language. Interestingly, the expected speed of execution of Tcl nowadays requires internal formats that are not just strings.

RUBYX11: this is not a language in itself, but many of the data structures in that library are organised as an extension of the type-representation structures already in Ruby: Class, Module, Struct, Object, and their existing conventions. RubyX11 adds type declarations and parametric types to an otherwise "weakly-typed" language, and yet, it doesn't even use them for type-checking (it may use them for type-checking, but that's optional, and the main use is something else: description of X11 packet formats). This library is actually one of the things that led to this paper.

1.2 A Common Conception of Types?

From this list you can see that there are many different conceptions of types and that they are not just extensions of each other: not every conception boils down to the same underlying idea. If you ask (or read) some people, they'll talk about their own approach as being *strong*, meaning that types are *obviously used because* of the need for validating. They'll refer to the other side as *weak*. The so-call *weak* side call themselves *dynamic*, meaning that types are *obviously used because* of the need for flexible dispatching. They'll refer to the other side as *static*. Neither side have only one use for typing, it's just that in each case there's binary opposition between validation and dispatching, two essential concepts of programming, and that each group shifts its own idea of normalcy and quality towards one task rather than another, regarding the other task as something that they'd rather not do[1]. (Meanwhile all of this is referred to as *static* by a third camp which call themselves *dynamic* meaning that a type is only a way of using data, that type is not part of the data or the container of the data.)

If we are trying to be all inclusive of those conceptions, we have to take the oppositions between ideas of normalcy and quality, that divide people, and consider each side as being one potential posture of the same person, and that those are to be used in different contexts because of their contextual advantages, stripped of their connotations. Politically, rejecting the connotations of both sides means being attacked by both sides (or hiding well). People don't tend to stick around in the no-man's-land, which is so called because in the midst of enemy fire from both sides, people don't tend to stick around. Polarisation of opinions thus encourages the further polarisation of opinions: people are encouraged to pick one side or *the* other. Now what happens if one refuses to pick any single one of the two (or three) sides? Then types can be used solely for what they can or could achieve in the structure of the program, and not for the purpose of bonding with people that adhere to the same cultural bias[2]. Naturally, this creates a new camp, the "middle side", or more accurately named, the "un-sided", and this divides people even more, but this further division is a necessary step towards deemphasising the divisions, although the goal here is not to force people to be more understanding of each other, it is just to develop the possibility of using a fuller conception of types than either side can provide.

[1]The *strong/static* camp is generally fine with dispatching as long as the type is known at compile-time: it is not fine with ad-hoc dispatching. The *dynamic/weak* camp (or part thereof) is so in favour of ad-hoc dispatching that not only it shuns type-checking, it shuns types themselves, unless they are directly part of the implementation (that is, when types are classes). Of course this doesn't apply to all members of each camp.

1.3 Towards An Alternate Understanding of Types

Here is a definition of types that should be reminiscent of a Design-by-Contract approach: a type is a set[3] of possible values that represents a set of expectations. A subtype is defined as a subset of possible values (this is called *covariant* because sub stays sub and super stays super). At the same time it is defined as a superset of expectations (this is called *contravariant* because sub and super are exchanged).

It's not like a type in a program would be represented as a data structure which gives direct access to every member of the type. The usual meaning of *set* as a data structure is something in which all elements are explicitly enumerated. It's usually not possible to think of a type as explicitly enumerated if the number of elements is unlimited in certain ways. A type is instead often represented as a property of an object (or of the content of an object)[4], usually called *the class of* that object, but more generally, it may also be represented by an expression which tests whether a certain object is a member or not, which may or may not be called *the class of* but surely *one of the types of*. Generally speaking, there's only one direct *class of* an object, for any object, and there can be superclasses of that class (desirably in a way that is linked to the implementation), but there can be unlimited types in a much more free way: as many types as there are ways for an expression to return *true* when given a certain value. For example, the number 36 could be considered as directly part of the *SmallInteger* class, and also as indirectly part of the *Integer class*, the *Numeric* class, the *Object* class, the *Comparable* class, and so on; but it is also part of types like *even numbers* or *multiples of 3*, or *square numbers*, or *divisors of 36000*, or more usually *numbers between 0 and 100*.

Here, class and object are generic concepts, where object can refer both to any t_pd or any t_atom; t_pd includes, for example, all objectboxes, all messageboxes, all floatboxes, etc.; whereas t_atom includes all floats, all symbols, all pointers, etc. This is important, because it allows to reuse the same type theory for the *t_atom kingdom* as

[2]Of course, once several people have the goal of being "type-neutral" and find each other, they bond, but the problem is not bonding per se :), it's its role in reinforcing cultural bias.

[3]Self-referential sets are forbidden in some type theories, thus requiring alternate definitions. Self-referential sequences of sets are allowed as long as they form a recurrence relation, and whenever they don't, it is possible that for some values you can't even *ask* whether it is member of the set or not. It's a matter of avoiding that an element is neither in a set nor not in that set, perhaps by preventing that sets may be defined in such a way. If this cannot be avoided, a 3-valued logic has to be used (in that case, the 3rd value is not *maybe*, it's *nonsense*, or in other words, the 3rd value is *can't answer this question, because it contradicts itself*, which is also sometimes called *mu*)

[4]You can define a set as being a property to be satisfied by its members, if that does not cause a self-reference.

for the *t_pd kingdom*. (In this paper, the word "struct" in only used in the C sense, and Pd's DS feature either fits in the t_pd kingdom or most likely might require a third kingdom).

What do we do with so many types? We use only those that we need. When do we need types? Maybe we need to ask ourselves when do we need classes? If following some classic OOP methodologies, classes might be lifted from nouns that appear in a natural language description of the project; but this does not address the fact that the need for naming classes may manifest itself differently in the programming language than the need for nouns in the natural language[5]. An "Extreme" methodology would follow the idea that classes are born as a result of refactoring classes that are individually too big (a mitosis-like process); in this case the program is less likely to contain classes that have little substance.

If the values of a type are mutable, then a difference has to be made between the set of all possible "objects" (meaning struct variables) and the set of all possible contents of those objects (immutable values aka *states* or *contents*). Then the set of all possible objects can be indefinitely extended by objects that could be distinguishing each other only by the fact that they occupy different places in memory, and that could be changing their own content as often as they want, so that the set of objects that have only one byte can be infinitely big instead of having just 256 elements. This is an example of why it is important to specify at which level we are thinking, in a given context, because types can have much subtler meaning than in usual programming languages. That type systems can apply to different kinds of entities (beyond the not-so-useful *kingdom* distinction described above) is an idea that I got from Cockburn[6] but he makes subtler distinctions than that. From distinctions between state types and object types we can talk about transitions in the evolution of the state of an object across calls, etc. But that ought to be a separate paper later on.

The organisation of types as they are used by a given program, is a sketch of a program. It is pseudo-code, in that it incompletely specifies what there is to do, but in another way than most other pseudo-code[6]; it is more abstract than what is usually called pseudo-code. That's the "interface declaration" side of pseudo-code (which is not usually called pseudo-code, but why shouldn't it?).

2. AUTOTESTING

2.1 Extreme Programming

In the creation and maintenance of software, a methodology is called *extreme* (or also *agile*) if it introduces feedback at all levels, so that requirements change according to changes in needs, specifications change according to changes in requirements, tests change according to changes in specifications, and programs change according to changes in tests. From a human point of view, testing is then something that precedes the actual programming, which can be done to the extent that tests can be performed by the computer, so that a human expends no effort in verifying that the software does what it should. This in turn lowers the cost of reprogramming any part of the software, which improves the flexibility of the software. Let us define *flexibility*, at the level of the programmer, as the amount of effort needed to modify the program such that it has the desired features, for every set of desired features. Extreme methodologies improve the speed at which new features may be added, because new features often require modification to existing code and those methodologies are designed to make that easy.

2.2 To Test is to Document

Automated testing programs is often considered as a form of documentation[**?**, **?**], and by some of those, as the only legitimate form of documentation of the implementation. In the latter case, it is said so because passive forms of documentation such as comments grow easily outdated, because there is not much of an incentive to keep it synchronised with the usual program. However, when automated testing is part of the development process (especially the test-first approach) the test suite can be considered as a form of documentation that *talks back to the developer* (as long as there's a *"Find Last Error..."* option and enough of an error message...).

There still is a need to distinguish between documentation made for testing and documentation made for understanding; the system of types can be a common part between those two kinds of documentation.

Unit tests are regularly considered as being *executable documentation*.

2.3 Ways of Testing

1. *Test-by-patience* is done by going through the manual and/or specification and verifying that the program conforms to it through human-computer interaction. No additional programs are used. This is a naïve way to test, as it is extremely time-consuming and either it costs a lot of money or numbs a lot of mind, most often both. However it's still the best way when no other way is applicable.

2. *Test-by-use* happens by trying to use the application in a realistic setting. *Beta-testing* is a special case of this. It is just as manual as *test-by-patience* but is less artificial so it makes one less willing to skip over it. It's *eating one's own dog food* (don't take it literally!). It is also more random, which is both an advantage and a disadvantage; but it tends to randomise in the direction of more usual situations.

3. *Test-by-rule (design-by-contract)* happens by writing code whose purpose is to describe what is true about a certain interface. For example, for a given method (message handler) there are valid inputs and there are invalid inputs. If the inputs are invalid (the *precondition* code reports *false*) then it's over, it's the sender's fault; else the validation continues after the execution

[5]This is related to *Execution in the Kingdom of Nouns[7]*

[6]All code is pseudo-code insofar as some things are not specified by the program and instead the interpreter or compiler has to fill in details. Some pseudo-code is significantly more *pseudo* than otherwise, because there doesn't exist an interpreter or compiler for it, or even further, because there couldn't exist one. There is another definition of pseudo-code which is a reasonably-specified language for which a compiler could be created but that was only created as a blend of other languages of the C/Pascal family as a way to reduce political tensions.

of the method, by checking the outputs. In a call-and-return system like nearly every OO language except pd, "the outputs" means the return value. In pd, this has to be interpreted slightly differently because the return has no value attached to it: any messages sent by outlets of the object during the handling of that particular message, count as being the outputs of that message. Dataflow means that instead of call-and-return it's call-and-more-calls. If those outputs are invalid (the *postcondition* code reports *false*) then it's the receiver's fault. If everything is valid, then either there is no fault or quite likely the rules aren't quite as thorough as they could be.

4. *Test-by-example* (unit tests, regression tests, integration tests) happens by writing code whose purpose is to probe a certain interface for the sole purpose of verifying whether it is correctly implemented or not. In the end, despite rivalry between some proponents of design-by-contract and unit-tests, and despite assumptions that one ought to choose one way or the other, design-by-contract reappears naturally when attempting to refactor unit-tests. When a rule covers the postconditions of several checks in a unit-test, and that rules are verified while the tests are running, then the tests don't need to check the postconditions anymore. Note that unit-tests don't have preconditions, they have particular examples that are presumed to be valid, and then postconditions that are particular results; or else they have particular examples known to be invalid, and they check that appropriate error messages are produced (but error-testing is not specific to *test-by-example*). In that sense, unit-tests are appropriate when rules are not known; they can be used as a way to figure out how to make rules out of examples, or more usually to avoid thinking about rules (people who can't stop thinking about patterns and rules will try the latter and do the former...).

5. *Test-by-proof* happens when a program examines the source code of another program looking for evidence that the program is working. The famous *Halting Problem* of Alan Turing shows that automated proofs are very difficult to do in the general case. There's no program that can determine whether a program will finish running or not, and answer yes or no, without risking not ever giving an answer; every candidate only can handle a specific level of complexity. The Halting Problem is not even one of the more complicated problems of its kind, because obviously, proving that a program gives the correct result, implies first proving that a program gives a result at all. In spite of that, there is some potential in making automated proofs about software (but I wouldn't know any pd user who would bother with that).

6. *Test-by-frugality* - DesireData's motto is "the best way to have less bugs, is to have less code". Not that by itself it is sufficient; nor that this is the only virtue of having less code, far from it: it is the "best" way because of its other advantages. Also related: "There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult."[7]

3. PATTERNS OF TESTING

3.1 *-test.pd aka [$1-test]

Those files are named in a way similar to help files, and they may share a lot with help files in terms of purpose, but they are abstractions that, when banged, make an automated test on a certain class of objects (or on a certain relationship of classes of objects). It outputs a list representing the result of the test, in a way that is easy to post-process and coalesce.

3.2 *-rule.pd aka [$1-rule]

Those files are contract-checking abstractions that wrap the original objects. For example, [moses 42] may be rewritten as [moses-rule 42] and this moses-rule will check the contract of moses and determined whether moses is being used properly (preconditions) and whether it is working properly (postconditions). It could also check invariants, which may be considered as a common postcondition to all possible input messages, or in paranoid mode as both that and a common precondition. The paranoid mode is only justified when there are elements outside of the control of the [$1-rule] object that can modify the state of the object.

In most implementations of contracts, the contract is considered to be part of the class, sometimes as being special sections in each method, sometimes as being method-like elements that are not real methods, and sometimes as ordinary methods in a method-combination context (e.g. CommonLISP, Dylan). This assumes that there are private (no-box) or protected (white-box) attributes that ought to be covered by the contract. This can be important for inheritance purposes. Pd doesn't support inheritance, so everything has to be done using composition (putting objects inside objects... that's what abstractions can do). This causes the protected (white-box) level to be non-existent.

The private attributes in Pd are difficult to monitor. Pd has means to refer to them using names sometimes, but most of them are so local to a specific patch part that it's difficult to use them outside of their normal context of use. This is one of the downsides of dataflow systems in general (and is corrected by introducing less *dataflowish* elements). The [v] object class is rarely used and can't do everything that float variables can. The send/receive attributes of [nbx] and [hsl] and so on, can't compensate for that, because you can't query the current state of a variable through a receive-symbol, without having all receivers of the send-symbol know about it.

Therefore it's generally better to focus on public attributes and black-box testing. Pd at this moment doesn't have full public attributes though: most attributes are write-only. Some externals support a "get" method but then the way to get the result of "get" to flow back exactly where it's needed ([demux] frenzy), as pure dataflow is about unidirectionality (returns have no values... *if* returns exist at all in your specific dataflow system)

3.3 Inheritance as Modified Composition

[7]This was said by C.A.R. Hoare.

Rule systems and test cases of several classes may share common parts. If those common parts are duplicated across classes, then this is a violation of the OAOO principle[13]. Therefore, some refactoring[14] ought to be applied, especially extracting common parts across [$1-test] and [$1-rule] files, making them [$1-test] and [$1-rule] files on their own. Those common parts are then reincluded in the original [$1-test] and [$1-rule] files as abstraction instances. Those common parts can exist independently of whether common parts exist in parallel in actual implementations; and there is potential for more complex hierarchies on the side of rules/tests than on the implementation side, as rules tend to be more granular than implementations, and because almost all current Pd classes aren't implemented using inheritance (most exceptions being related to Flext, PyExt, GridFlow).

Thus we can get a lot of the effect of inheritance by using composition instead, which is a realisation that is quite in vogue in the OOP world, especially after the initial abuse of the inheritance feature. However, there are (at least) three forces that pull us back to inheritance:

1. The need to share a same self. In Pd the self is called "$0". When there is anything special that ought to be shared among the wrapper object and the wrapped object (with possibly more than two levels of them), there is incentive in merging the selves so that several different abstraction instances have the same $0.

2. The need for the wrapped object to use methods overridden by the wrapping object. This is a useful way to make objects more configurable. However, it is harder to come up with a way of actually implementing this in pd, and I don't see this particular feature as being particularly necessary for testing purposes (at the time of this writing...).

3. The need to handle virtual inheritance. This is when you have a diamond inheritance (B and C are subclasses of A; D is a subclass of both B and C) but repeated superclasses do not cause repeated subinstances (a [D] object does not include two different [A] instances, just one).

3.4 Linearised Multiple Inheritance

This is the kind of multiple inheritance present that is native to CommonLISP, Dylan, Ruby, etc. and that was new in Python 2.2 and Perl 5.8. AFAIK they all use basically the same ordering except Python 2.3 and later. This is the way of supporting virtual inheritance at the level of method-calling. C++ supports virtual inheritance for object attributes, but not for methods; it is because it has no lineariser. The way it works in those languages is that there is a pseudo-method called "super" that is actually the next method of the same name, as found by the lineariser. Because Pd's abstractions don't have the concept of method (it has to be reinvented using [route]/[list] combinations), what is actually appropriate is to fake nesting of abstraction instances (but reusing the same $0) using a pseudo-object called [super] which would have the appropriate number of inlets/outlets.

In practice, this is a feature that is trickier to implement than others present in this paper, so its actual implementation will be delayed. The main effect of doing this, is that some tests and rules will be checked twice or more for the same object class, if that class has inheritance complex enough that it contains at least a diamond (of some shape: what counts is that there is more than one path to the same superclass; then without linearisation, all paths will be explored).

3.5 Data Type Segregation Causes Rigidity and Duplication

An object (abstraction or not) can choose each of its inlets and outlets to be either message-oriented or signal-oriented. It has to choose one or the other. Because of this, [+-test] and [+~-test] have to be completely segregated, and yet the contents of [+-test] would be much similar to the contents of [+~-test], in the same way that [*-test] would be like [*~-test] and so on; there exists a specific way in which one can make a patch for a [$2~-test] from the corresponding [$2-test] patch. (I say $2 here to avoid confusion with the upcoming $1)

3.6 Parametrised Class Names With Separate Definitions

Suppose every difference between messages and signals is encapsulated as a pair of objects: e.g. [t~] pretends to be a [t] but is just a dummy for the purpose of implementing something signal-related in a message-or-signal context (that is, a context in which you don't know in advance which is which). If every message-vs-signal difference becomes only one character away in the name, then you can access either using [t$1] where $1 is the type of connection (message vs signal) that you want to handle. Because of the difficulty of writing down the empty symbol in pd (supposing a non-modified pd syntax[8]), it might be better to use explicit suffixes for each of those connection types that has to be supported. Because most of the similarity between messages and signals occurs with float messages, the actual suffixes in use are: float context [$2.f]; signal context [$2.~]; grid context [$2.#][9]. Those suffixes become patch parameters like [t.$1]. A period is introduced to separate the suffix from the root name because of ambiguities arising when suffixes can also be letters.

3.7 Parametrised Class Names With Common Definitions

Once diverging elements between similar abstractions have been abstracted out, those similar abstractions can be re-expressed as one abstraction in which [t.$1] and [swap.$1] and [inlet.$1] and [outlet.$1] are in use. This abstraction does not actually uses the [$2.$1] suffixes and instead is used as [$2 $1], that is, a separate argument. The name of a class is usually assumed to be a single symbol, but even in pd's builtin classes, [list] is actually several classes whose constructor names are symbol pairs[10] and for which each

[8]

[9] in practice, GridFlow messages and objects still have to be treated separately from ordinary messages and objects, and this will continue for some time, even though a core design principle of GridFlow was to make grids look like an extension of the existing semantics of atom messages of Pd (or originally of FTS). Further separation of # into #b #s #i #l #f #d will happen later due to the several number systems of GridFlow.

[10] a similar concept called *ensemble* exists in Tcl, which also has several namespace systems: the other one is based on

corresponding class name is a symbol made of those two symbols joined by a space, e.g. a [list prepend] constructor with a [list\ prepend] class name. I don't call the dot notation a namespace system because it doesn't have any import/export options so it's not much different from using underscores, camelcase or plain concat.

Equivalence between dual-symbol and single-symbol notation can be achieved in several ways. Using abstractions, you may create a [$2] abstraction which takes a $1 argument and only contains a [$2.$1] object surrounded by [inlet.$1] and [outlet.$1]. In this example, note that $2 is a constant substituted manually whereas $1 is a true pd variable. To get the other effect, you need to create one [$2.$1] object for each possible combination of $2 and $1 and put a [$2 $1] object inside. In this example, note that both $2 and $1 are constants substituted manually.

Doing it with abstractions is not only boring, it is also troublesome: for example, when it comes to variable number of inlets and variable number of outlets. In this case, achieving the effect of simple aliasing can be done using externals that don't create new classes, but instead create aliases or alternate constructors that act like aliases. That kind of technique is already required for defining [inlet.$1] and [outlet.$1] themselves.

4. PRACTICAL TYPE HIERARCHY

4.1 PackLike and UnpackLike

Many object interfaces in Pd can be thought of, in a simplified but accurate manner, as being an object sequence of [pack] -> [foo] or [foo] -> [unpack] or even [pack] -> [foo] -> [unpack]. Thus it is possible to define PackLike as being the requirement that left inlet is active, has a "set" and a "bang" method; other inlets are passive; and values accepted are atoms. UnpackLike may be defined as outputting only individual atoms with a right-to-left order. Those are characteristics of many classes that are also non-characteristics of many other classes, and it is most effective to document those behaviours using shortcuts like PackLike and UnpackLike than spelling it out each time.

4.2 TotalOrder

The behaviour of the relational operators [==], [!=], [<], [>], [<=], [>=] forms what is called a TotalOrder. The [==] forms an Equivalence, which is Transitive, Reflexive, and Symmetric (Commutative). [!=] is Symmetric but otherwise opposite of [==]. The other operators form the TotalOrder proper. They are Transitive, AntiSymmetric (AntiCommutative), and either Reflexive or AntiReflexive depending on whether they include or exclude equality. The names of those properties are standard from Algebra books or are guessable variations on standard names. Those rules, as stated in math books, are readily translatable into rule files.

4.3 Commutators vs Commutative (-tor vs -tive)

In algebra, the commutator of a and b in the context of multiplication, is $ab - ba$. By comparing this with 0 you can figure out whether an operator is commutative about certain possible inputs: $ab - ba = 0$ is the same as $ab = ba$. Similar constructions can be made for other properties, yielding associators like $(ab)c - a(bc)$, invertors like $ab/b - a$, distributors like $ab + ac - a(b + c)$. Those new *operators about operators* evaluate how much Pd (and the CPU) respect the basic assumptions made in everyday math. If this seems futile, then think about other more complex operations (on colours, points, shapes, images, etc) that recycle those basic assumptions in their respective contexts.

double-colon prefixes and infixes.

4.4 Approximative Algebra

The factorisation of each algebraic property into a new operator and a = 0 isn't just made for saving an = 0 at the end: this is actually an opportunity to replace the = 0 by something else representing tolerance to error. Float operations cannot be assumed to be exact (yet in certain patterns they will be exact). Think of = 0 as being a *closed ball of radius 0* (this means: check that the norm of the result is ≤ 0). Replacing the 0 by an actual error margin you get another closed ball which consists of all values considered to be sufficiently similar to the exact result. A *norm* is something that computes distances in a translation-independent way: thus it gives the distance between y and x by computing the distance between $y - x$ and 0 because translation-independency guarantees that subtracting x on both sides preserves the result. (Norm is thought of as a single-input operation). For floats, [abs] is an appropriate norm, whereas for signals, [env~] (aka RMS) is the best equivalent.

This allows appropriate balance of precision requirements and memory requirements: rounding occurs mostly because floats use a fixed amount of memory. Approximative situations happen because floats are used where Real Numbers are assumed, among other things. Comparing results in "fuzzy" ways (not necessarily in the same way as the thing officially named Fuzzy Logic) allow to pretend that the rules are hard in the way that they are written but to still apply them in a sufficiently lenient way with well-controlled error margins.

4.5 More

This is a work-in-progress for which some additional documentation is available (e.g. about stochastic testing, worst-case testing, benchmarking, alternative norms, error message redirection, testing for appropriate error messages, etc.). For more information, see the PureUnity and DesireData projects.

5. REFERENCES

[1] The PureData Documentation Project (PDDP)

[2] RubyX11, Mathieu Bouchard, 2001. Shown at Europäische Ruby Konferenz (EuRuKo) in 2004 as a paperless presentation. The actual software is downloadable from http://artengine.ca/matju/RubyX11/

[3] Kent Beck and Cynthia Andres - 2nd edition of *Extreme Programming Explained* - Addison-Wesley 2005

[4] Erich Gamma - foreword to 2nd edition of *Extreme Programming Explained* - Addison-Wesley 2005

[5] Richard P Gabriel - design patterns

[6] Alistair Cockburn - A Constructive Deconstruction of Subtyping

[7] *Execution in the Kingdom of Nouns* from Stevey's Blog Rants,

http://steve-yegge.blogspot.com/2006/03/execution-in-kingdom-of-nouns.html
[8] http://en.wikipedia.org/wiki/Unit_testing
[9] http://www.ibm.com/developerworks/library/j-test.html
[10] http://www.ibm.com/developerworks/java/library/j-diag0814.html
[11] http://developer.apple.com/tools/unittest.html
[12] Barbara Liskov - The Liskov Substitution Principle
[13] Portland Pattern Repository - *OnceAndOnlyOnce* - WikiWikiWeb at http://c2.com/cgi/wiki?OnceAndOnlyOnce
[14] Portland Pattern Repository - *ReFactoring* - WikiWikiWeb at http://c2.com/cgi/wiki?WhatIsRefactoring
[15] Flext
[16] Pyext
[17] GridFlow
[18] PureUnity
[19] DesireData