On The End of TIME and BEING

Why are Grades not Types?

A question that turns up frequently in framework conversations is, "Why are Grades not Types?". At the outset of development, this question wasn't completely clear, although it was clear enough that we needed to stake out some new territory to justify choosing a fresh term that wasn't loaded down with all the connotations of decades of computer science baggage. Initially this choice was intended as an aid to thought, rather than necessarily crystallising a well-defined meaning. However, in the couple of years since that point, a number of crucial and reasonably clear-cut aspects of the distinction have emerged, which this page will describe. We expect these distinctions will become more and more refined over time, and the relationships between their different aspects will become more and more clear. We argue below that the traditional Computer Science concept of "type" is very deeply flawed at the practical and pragmatic as well as cognitive and philosophical levels, and that traditional type theory fails to provide a basis for delivering software that can accommodate an open ecology of contributors—developers and users alike.

Before delving into details, it's worth laying out a few practical and pragmatic aspects of what grades are so that at least some of the dimensions of the elephant can be felt. A grade "is" a JSON structure that is assigned a stable, public name in a hierarchical namespace. Currently a grade is defined in a live Infusion system using *fluid.defaults*. Here is a small but reasonably representative example:

```
    fluid.defaults("fluid.prefs.panel", {
        gradeNames: ["fluid.rendererComponent", "fluid.prefs.stringBundle", "fluid.prefs.modelRelay",
"autoInit"],
        events: {
            onDomBind: null
        },
        listeners: {
            "onCreate.onDomBind": "{that}.events.onDomBind"
        }
    });
```

In this case, the name of the grade is **`fluid.prefs.panel`**. The JSON defining it begins on the { character on the first line, and it starts by listing a collection of ***parent grades*** on the second line. It continues by listing various other piece of JSON material whose meaning is governed by the parent grades mentioned, and that of further ancestors, etc.

The principal kinds of grades we will discuss here will be ***component grades***, which are grades somewhere in their hierarchy derived from **`fluid.littleComponent`** which is the current root component grade in the Infusion system. Other grades not representing components can represent more primitive entities such as functions, or even more simply, any arbitrary, hierarchically-resolved block of JSON-formatted metadata of any kind. It's a crucial feature of component grades that they will tend to mention in their definition certain strings formatted in a particular way - beginning with a section delimited by { } braces named the ***context***, and following with a set of ***path segments***. For historical reasons, these paths have been named ***EL expressions***. Exactly how these expressions are interpreted and resolved is key to the discussion which follows - and indeed it's fair to say that this resolution process is crucial to the meaning of grades (vs types) as we will be discussing them. In addition, although the basic meaning and function of this resolution process has been fixed, the exact mechanism is continuing to undergo subtle enhancements which increase the user's "power of reference".

Background on terminology

Confusingly the word "type" appears in some places in subcomponent definitions, as well as the word "typeName" labelling the finally instantiated component. Before Infusion 2.0 it might be worth rationalising all of this to simply use the word "gradeName" and replace the field "gradeNames" above with the field "parentGrades" or so.

With its resolved hierarchy and appearance of traditional inheritance, this all looks as if it might boil down to some kind of classic type system, so we will spend some time discussing the various important aspects to the question of why it doesn't, and indeed mustn't.

Some reference pages about Infusion's grade system and EL expressions:

- Component Grades
- EL References

The Key to Time

One of the facilities that has emerged in our implementation of grades is the ability for the users (i.e. implementers, developers, and extenders at large) of a component to adorn it with further grades "after the fact". However this condition of being "after the fact" requires quite some unpacking due to the very different nature to the passage of time during component instantiation. In this case, the "afterness" we are talking about is "after the fact of definition of the original component", as well as "after the fact that a particular grade has been selected for use in a particular context". Wherever she stands in the *chain of creators*, the final consumer of the product of a grade definition always has the chance to contribute further grades into it.

In a system with ordinary object instantiation, this would usually imply that these kinds of "after the fact" also would imply the further "after the fact of the beginning of the instantiation process of the object (component) in question". This correlation of the different kinds of "after the fact" led us to consider that this paper - "Three Approaches to Object Evolution" (Google Israel Engineering Center, Cohen and Gil, PPPJ 2009) - described a system similar to ours in relevant ways. This describes a object evolution system implemented in Java - which gives users the powerful capability to decorate an already created object instance by granting further types to it, and explores ways in which such a system can be made safe and more predictable by the adoption of various constraints such as *monotonicity*, which is a form of extension that only allows evolution to a subclass of the object's current type. In Java, such a system is ambitious and peculiar, although in other languages such as the prototypal family stemming from Self (the native object system of JavaScript itself is derived directly from this family) such "type morphing" is standard and clearly provided for. Another example of strands of type-evolving objects lies in numerical algebra systems such as IBM's (formerly) Axiom, and, presumably, the much more popular Mathematica.

However, despite allowing the apparent affordances of type morphing, our system is not one of this kind. In fact, the grade content of an instantiated object is completely fixed after its instantiation point. This kind of immutability seems quite desirable in terms of being able to reason clearly about the behaviour and content of objects. This is especially the case since, as we will shortly see, we give up so many of the other pieces of type machinery that appear to be motivated by the aim of facilitating reasoning, not to say theorem proving, about the behaviour of objects. It's worth looking in some detail at how our different conception of **time** allows us to let one of time's arrows to point in the opposite direction to the other two.

We will present here a couple of axioms which it can be "discovered" underlie all "type" or "object"-based systems that we are aware of, which are helpful in drawing the distinction between our model of *instantiation work* and that of others.

Infusion's Principles in Comparison to Object Orientation

The Object-Oriented Theorist's Hidden Axiom of Orthogonal Time

If an object of type A is composed using a subobject of type B (whether through inheritance, aggregation, or some other scheme) there must exist two definite points in time in the lifecycle of A i) when the B subobject starts to construct, and ii) when the B subobject finishes construction, and between which times no work is done on A that is unrelated to B.

The Object-Oriented Theorist's Hidden Axiom of Atomic Delivery:

If an object of type A is composed using a subobject of type B, from the point of view of object A, the B subobject is delivered in an apparently atomic process into which the implementation of A has no insight.The B subobject is delivered in a fully initialised condition, having previously been seen to be entirely nonexistent.

From the point of view of these axioms, the two extremes of the object-based systems we describe above now look quite similar. Whether the addition and removal of subobjects/capabilities occurs early—during the overall construction process (as in traditional OO such as C++ or Java)—or whether it occurs late in the life of an object (as in Self or Axiom) it still occurs atomically, and in a way which, without necessarily bringing in a particular model for asynchrony or parallel processes, could be described as "synchronous".

In Infusion's instantiation model, we give up the conception of an "object" as being enclosed within rigid boundaries, and also abandon the object theorist's horror of partially initialised objects. Every text on the subject is clear that we must absolutely avoid them; a partially initialised object is considered by language lawyers to be a source of "undefined behaviour" and must not be observed by outsiders. Instead, Infusion adopts the capability of seeing an entire tree of instantiating objects as an undifferentiated whole, whilst from one point of view still capable of being carved up into separate objects. Instantiation breaks as a wave across the entire tree as one process, and could in theory pass from evaluating ("observing") any one object property to any other. In exchange for this loosening of restrictions, we adopt a powerful constraint in order to compensate.

The Infusion IoC Core Principle of Immutability

Any property which has been observed on any object during the instantiation process may never be observed later in the process to hold a different value.

This in some ways can be seen as closely parallel to the functional programmer's notion of *referential transparency.* That is, that the value of a symbol, once bound, can never be rebound, and any complex expression can be replaced by what would be its fully evaluated value without changing the meaning of any wider expression in which it is embedded. However, there are some very important differences between these principles. To a functionalist, the names he uses to denote things are *private* and are only endowed with meaning within a particular (local) context. In contrast, the names we use in Infusion IoC are *public* and have stable referents within a global context—usually the widest referential context which can be operated by whatever virtual machine technology is in use (for example a browser frame, or a VM context).

Again, we operate an exchange in order to compensate for this potentially destabilising expansion of capability. Whilst we insist that all state be addressible within a global context, we greatly restrict the observational power that can be used to inspect this state. Whilst in theory it is available to be inspected by all running code, in practice we perform all observation through the means of EL references. This shows another underlying principle at work, which is again wholly at odds with the mentality of type and OO theorists:

The Underlying Infusion IoC Core Principle of Enablement

The system will assume that every citizen is of good intent, and make no efforts to forbid undesirable behaviours, but instead concentrate maximum efforts to promote the positive effects of desirable behaviours.

Many type systems make a lot of fuss about prohibiting "crimes," but in practice it is commonly understood that the determined criminal can never be effectively thwarted. For example, this article by Steve Yegge - http://steve-yegge.blogspot.com/2010/07/wikileaks-to-leak-5000-open-source-java.html - appears to be satirical, but makes a pretty straightforward point that the removal of all access restrictions from every Java project would vastly increase their value to the community at large. Unfortunately such instances of observing the emperor's lack of clothes have no effect on the development community, who have spent many years training themselves and each other to neglect the force of such arguments. Another anecdote in this tradition is "#define private public", discussed in this stackoverflow posting: accessing private members. This has led to an amusing "arms race" between developers and compiler authors as shown by the following report that Visual Studio 11 explicitly tries to plug a few such holes: Farewell to #define private public. These sorts of discussion should make it clear that protections at the level of a type system were never intended to provide "security" even though many of those involved behave as if they do.

However, whilst on the one hand providing no guarantees of real safety, such protection systems do on the other hand have an important damaging effect to the community - they increase the likelihood of forks in a project where the communities cannot agree on what system elements should be exposed and which not.

We take a different view - the problem of "appropriate access" is one of *vision* and not one of *enforcement*. This statement needs a bit of unpacking - we will talk around it in the next few paragraphs. What we intend to take away from the above examples is that the approach to access and modification of state in traditional type systems is fundamentally broken, if it allows disputes like these to arise. We will simplify some of the resulting discussion by taking at least one leaf out of the book of functional programmers. As well as the analogical immutability (access to globally addressed state seen as analogical to "referential transparency") that we adopt in the "core principle" above, we also recognise the value in the real thing - that is, the value of programming as much as possible with ***pure functions***. Now - if a pure function has no side effect on any part of the system, how could it be that there could be anything other than positive value in exposing it for use to all users of the system? That is, how could it be that exposing every pure function defined with a system for universal public use represents a "danger"?

The only such danger could lie with respect to changes in the system. If a pure function known by a particular name in the system changes its implementation in a future version to compute a different function, the user's call site will break. But this is surely a problem of communication, rather than of function. The implementor of a function should be expected to advertise his level of commitment to the stability of implementation - and in the absence of such an advertisement, the user should use the function with full awareness of the risk that their call may be broken by an update. However, to make the decision that "the risks of communication are too great" and to react in the first instance to make all such functions inaccessible seems not only cowardly, but even sociopathic - it is an acknowledgement that the risks of communication seem so great that the only rational response is to agree to provide no value - that is, to enter into no relationship at all with a community unless there are guarantees of a positive outcome.

When seen in communities of normal human beings (that is, those other than software developers) this sociopathy is easy to characterise, and all of the standard human responses (increasing the level of social contact, mutual understanding, making attempts to build truest, etc.) are appropriate. However, in the community of software developers this sociopathy is universally accepted not only as the norm, but even the touchstone of virtuous behaviour.

However, this model of sociopathy runs much deeper than merely the world of software developers. Richard Rorty, in his 1994 essay "Ethics Without Principles"[1] explains - " ... the central flaw in much traditional moral philosophy has been the myth of the self as nonrelational, as capable of existing independently of any concern for others, as a cold psychopath needing to be constrained to take account of other people's needs. This is the picture of self which philosophers since Plato have expressed ... " and later on decries " ... the sort of person envisaged by decision theory, someone whose identity is constituted by 'preference rankings' rather than by fellow feeling." Clearly this description doesn't fit the reality of the vast majority of existent people - even software developers. However, a culture which reinforces and rewards such descriptions, and thrives by mutual projection of such descriptions onto the self and others can go a long way to creating the *impression* that this description does fit.

To return from this deviation into moral philosophy, we insist that the universal, and maximally intelligible, advertisement of all artifacts held within a piece of software to the public is obviously desirable, and will now talk about the means we use to ensure that this advertisement is as productive and safe as possible.

Firstly, we are protected from the effects of advertising functions, largely by making sure such functions are *pure*. Now we will go on to explain how we plan to be protected from the effects of advertising *state*. The dangers of global access to state lie primarily in uncoordinated access from multiple sites, engaged in a race to modify the same piece of state. The meaning of "uncoordinated" should be unpacked - in a traditional programming language, one could say that this access is "unannounced" - the reference occurs from a private site in the middle of a function body, and the capability of a function to perform this access does not need to be (and in general *cannot* be, due to a lack of primitives in the programming language) announced to the outside world. This, for a start, seems a crucial failing in the type theories which promise to prove that the use of particular artifacts is "safe". We will criticise these kinds of theories later on when we consider "Liskovianism", commonly held as the bedrock for a large family of communities offering type theories. However, we'll note here that some other communities, faced with this problem, adopt a very different solution - certain "pure functional" languages (whose ancestry was rooted in the originally "impure" ML but leading to such "pure" descendants such as Miranda and Haskell) forbid any kind of destructive updates in the system whatever.

Preserving the limited appearance of immutability

We step back from such an extreme solution - even the proponents of such languages admit that their properties are hard to reason about, and furthermore they are a poor fit for the world as experienced by humans which arguably consists entirely of state (some philosophers disagree). Our compromise solution consists of two parts:

1. That the "well-known location" of the state holding the arguments and results of the application of every function (expected to be pure) in the system is advertised as part of the application record (these records are held within *grades* which finally starts to close the circle on this long discussion). Such records primarily take the form of Invoker and Listener records.
2. That update to state in the system proceeds in "fits" (perhaps, in a more recognisable term, "transactions") in which certain rules strongly restrict the kinds of updates which can be operated or perceived. We recognise two main varieties of "fit":
    a. A "component initialisation fit" in which all observers agree that any piece of referenced state participates in just a single update after which point it is considered immutable - this could be considered a form of Single assigment.
    b. A "model update fit" in which *managed change* occurs to a piece of *model material* (these to be defined later). In this form of fit, external observers (that is, those pure functions implementing the body of a component) observe no change in state until the fit is completely resolved. At this point the update appears to execute atomically, although observers within the fit (primarily *model transformation functions* comprising *lenses* or other less information-preserving mappings) appear to see the update as composite. This distinction between observers inside and outside the fit allows us to work with updates that appear composite and complex (and perhaps distributed over the component tree) from one point of view, whilst appearing simple and atomic from another. This lets us head off the risk of many kinds of races and corrupted state. Some discussion on this is in New Notes on the ChangeApplier,

Related to the discussion in 2a and 2b is the possibility for destruction and reinitialisation of components. The destruction process is considered "externally atomic" in the style of 2b update fits, even though it is clearly internally a composite process. SInce all access to state is mediated by the contents of application records, the system can clearly distinguish between the two kinds of observers. Since every function is individually "morally pure" (in real-life code we permit the actuality of purity to be suspended, as long as the "intention of purity" is maintained - similar to the scheme in which functions declared `const` in C++ may actually modify state, usually to maintain caches, etc. in the interests of efficiently maintaining an external appearance of mutability), there is no possibility for any individual observer to "observe" an update, even though the collection of them in their entirety has an update propagated through them.

From the "framework's-eye view" (and also from the view of those users and developers constructing, observing, and reasoning about the application) there is a clear appearance of state in a conventionally addressed and indexed form, whereas from the point of view of any individual implementing (pure) function, there is not.

With some of the pieces in place regarding the "normalcy" of advertising to the framework the locations of referenced state, we can briefly return to our "moral view" of the social process by which developers collaborate. Since this style of "advertised access" becomes an expected convention for code participating in the framework, any code not participating in this convention stands out. It may be that the code is failing to follow the convention for "moral reasons" (for example, the kinds of situation referred to earlier as "maintaining the intention of purity" whilst operating mutable caches of frequently-used or hard-to-compute state) or they may be "criminals" - that is, they are operating direct access to state for reasons which are not justifiable in terms of maintaining the integrity of the overall application. In either case, the violation of the convention is rare, and flags to the reader the code for special attention and careful reading. As it passes through a community review process, we expect such code to be either passed or rejected (by an individual or a distributed process of approval) - but the fact that it is rare and easily visually distinguishable, we argue, provides all the "protection" that we require. Our different view of the real moral intentions of developers (that is, that they are not in fact "cold psychopaths needing to be constrained", but really in fact "relational beings whose intentions need to be advertised") suggest that all that is really required is a scheme for the intentions of developers to become clearer.

Choosing the right abstractions

It may be that the most important difference in our approach is not the particular formalisms that we arrive at, but more the kinds of methods we would prefer to reach them by. We've surveyed a few other solutions to the problem of managing *state* and *behaviour*, and before we reject some more of them, it might be useful to try to characterise some more aspects of the mentality they spring from. In Computer Science, and the wider philosophical tradition it stems from, there is a high value placed on constructing *consistent formalisms* that it is easy to prove theorems about. This consistency and formal tractability often ends up being privileged to the detriment of ensuring that the formalisms are relevant to any particular purpose. Thomas Kuhn, in *The Structure of Scientific Revolutions* [2], divides scientific progress into two kinds of phases. The first of these he labels "Normal Science", which is a period of puzzle-solving, where known intellectual rules are used to push around problems much in the scheme of solving a crossword puzzle. The second phase he labels "Changes in World View" which he explains are "not fully reducible to a reinterpretation of individual and stable data". Our manifest failure as a profession to arrive at consistent engineering standards which allow us to rapidly and successfully solve new problems for our users should convince us that we are not currently in a position of "normal science" - therefore we should treat with suspicion techniques which are selected for the efficacy in solving problems in that regime. Theorem proving may well come later, but we should first accept as a profession that we still simply have no idea what we are doing.

In order to understand what we are doing, and find out which abstractions are appropriate for the tasks we are interested in, we have no alternative but to try to study ourselves as we are about the work of solving real problems. This can't be done as an "armchair exercise" - we must have real work to do, and we must also have the spare capacity in order to observe ourselves over an extended period, and evaluate the success or failure of different kinds of primitives and abstractions in successfully and economically explaining the kinds of work we find ourselves doing. For different reasons, this process is very difficult to conduct in either industry or academia as they are constituted today, but with a lot of patience, it can be slowly conducted at least somewhere.

The Bankruptcy of Liskovianism

It's time to take on one of the the most important formalisms underlying type theory as it is most broadly practiced by software engineers today. The Liskov Substitution Principle [3] is summarised at Wikipedia as follows - "If S is a subtype of T, then objects of type T may be replaced with objects of type S, without altering ***any of the desirable properties*** of that program (correctness, task performed, etc.)" (italics ours). On the face of it this definition appears so absurdly broad and self-defeating that one has to imagine that it must have been misquoted. However, going to the original paper one discovers this wording: "the objects of the subtype ought to behave the same as those of the supertype ***as far as anyone or any program using supertype objects can tell***" (again italics ours). This original formulation is even more incoherent and self-defeating - if the objects behaved the same "as far as anyone can tell" what would it mean to say they were different at all - and what purpose would there be in even having a subtype? The paper then quickly dives into formalisms such as "An assignment x: T:= E is legal provided the type of the expression E is a subtype of the declared type T of variable x" or else "Let q(x) be a property provable about objects x of type T, Then q(y) should be provable for objects y of type S where S is a subtype of T". All this occurs without a good attempt to motivate what kinds of purposes this definition is meant to be good for - the "statement of intent" is so grandiose as to make the mere existence of subtypes completely impossible.

The treatments do calm down after a while and talk about the kinds of properties that can in fact be handled by practical implementations supporting Liskov subtyping. Unfortunately these are all rather boring kinds of "bean-counting" properties such as type assignability and agreement of signatures - the approach explicitly disclaims being able to handle vital considerations such as usage of computation or space, or even whether a computation will terminate at all - or, for example, whether an instance of a type can be serialised in a straightforward way with some bounds placed on the storage used. Liskov94 even states "We are interested only in *safety* properties ("nothing bad happens")". This should be a warning to us based on the "Principle of Enablement" described above - this entire philosophy stems from an attempt to minimise ill consequences and not to maximise beneficial ones, and that is all it ever set out to do from the start.

However, there are numerous examples even of seriously ill consequences that this approach fails to rule out[4]. Here are two presentations of an "object" that appear to satisfy the same type contract -

```
// Implementation A: (Good Liskov)

public class ImplA {
    public boolean value;
    }

// Implementation B: (Bad Liskov)

public class ImplB {
   private Queue<Boolean> history;
   private boolean toggle = false;
   public ImplB() {
       history = new ArrayDeque<Boolean>();
       history.add(false);
   }
   public void setValue(boolean b) {
       history.add(b);
       toggle = !toggle;
       if (toggle) {
           history.remove();
           }
       }
   public boolean getValue() {
       return history.peek();
       }
   }
}
```

This is written in near-Java - note that due to Java's odd approach to "accessors" we can't cleanly write `ImplA` in a way that shows that its type contract is indeed identical to that of `ImplB`, but in other languages (e.g. Python) we could. Now it's clear that a description of the contracts of these objects of the form "They expose a method which can be used to set and get a boolean value" couldn't distinguish between them. But Liskov is clear that the LSP is more than merely about type contracts - the description of the principle talks about "any property provable". But where are we to get these properties and their proofs from? They clearly don't lie in the target language or indeed in anything which could be directly tied to it. Assume that we were seized with inspiration, and actually set about trying to verify the property "Both `ImplA` and `ImplB`'s accessors will always return in a finite time". If we were heroic, we might be able to succeed in proving this property - but where would we seize upon the formalism to do it? And had we succeeded, we would still not achieve the presumably useful goal of informing the user of `ImplB` that their implementation has a property which would be surely destabilising to any architecture of which it was a part - it consumes memory without bound when in use, and could never expect to have its entire state serialized to disk with bounded use of storage.

It's clear that the LSP is only capable of checking the most harmless and unilluminating properties of implementations without forcing the reader to go to the source code looking for inspiration about the kinds of "properties" he is interested in proving about them. Our aim, though, as we explained above, is to promote the implementor's power of "advertisement" about their intention, especially where those relate to the handling of updates to state. This is precisely at odds with the intent of the LSP - and indeed furthermore those of both functional and object-oriented programming as a whole - which have as a core motivation promoting the developer's ability to ***hide*** the nature and location of updates to state, rather than to advertise them. This is, therefore, one of many reasons that we abandon the affordances of the LSP in our system, and this becomes one of the crucial answers to our headline question of "Why are Grades not Types?" - because in fact we explicitly abandon the crucial definitional crux of what a type *is* to most practicing communities. We do not say, given that grade S is a parent grade of grade T, that we make ***any guarantees whatever*** on the behaviour of T instances based on this relationship. And in fact, another of our core principles:

The Infusion IoC Core Principle of Open Creation

Any decision made about an implementation by one author in a system can be reversed by any other author who is making use of it.

positively forces us to avoid making such guarantees. All that the system guarantees is that a particular algorithm is used to combine grades together - the meaning of the resulting structure is entirely an issue for the user who designated it to come into existence[5]. However, we try to "rig" the system for interpreting the contents of grades in order to ensure that at least i) every such resultant merge is the expression of *some* valid structure, and ii) raise the chances that the merge expresses the plausible intentions of *some* creator - by choosing "appropriate" primitives and appropriate ways of aligning their addresses in the structure. This can only be done by appeal to our actual experience and our success in predicting our own intentions (see section above). It can't be done just through the exercise of reason - that is, it is "true science" as opposed to what Kuhn identifies as "normal science". It also falls into the category of what Lakoff would describe as "neither wholly predictable nor wholly arbitrary" - that is, it amounts to a language structured by metaphorical relations.

More about the tradition that Liskovianism stems from

If Liskovianism is so unhelpful, why is it so popular? Part of the answer is found in our previous discussion. Formalisms like the LSP are popular because they provide a fertile ground for easily proving theorems about implementations, and thus perpetuate the delusion that Computer Science has reached Kuhn's "crossword-puzzle solving" phase of its mature life. However, as we have argued, the kinds of theorems that we can prove are seldom about things that help us do a better job for our users. A further clue about the attractiveness of the LSP can be found in its description for "Model of Computation" (Section 3, p 1816): "We assume a set of all potentially existing objects, *Obj* partitioned into disjoint typed sets. Each object has a unique identity. A *type* defines a set of *values* for an object and a set of *methods* that provide the only means to manipulate that object". This is a view of the world that stems particularly from a strand of Western philosophy that can be traced back to classical Greek thinking and whose stream flows through such thinkers as Descartes, Kant, Frege, Chomsky and numerous active living proponents. However, this is a view of the world that has been conclusively refuted as a good match for how human cognition actually works[6] - that is, we do not in fact model the world as consisting of *entities* which participate in *relations* and which can be exhaustively divided into a set of properly nested *types* based on constraints on their observable *properties*. Now, if this is a poor model for the cognition of our users (not to say, also ourselves), it is hardly likely that it will be the key to a effective and efficient way of making artifacts that work well for them.

It seems no accident that the examples which are used to motivate Liskovian substitution are highly contrived ones drawn from the domain of computer science itself. For example - Liskov94 has as a central example two data structures, a *bounded bag* and a *bounded stack* and spends time showing that, in terms of the principle, one is a subtype of the other. This is reminiscent of the "peculiarly well-defined examples" such as "cats sitting on mats" which Lakoff pillories generations of philosophers for appealing to in their push for a property-based decomposition of the world. The point is that such decompositions when they are "peculiarly successful" in this way actually rely on a wealth of accumulated experience in a domain which has *already guided* us, the reasoners, to come up with the right distinctions in the first place - the choice of these examples make the formalism and the manipulations that it relies on seem inexorably correct. In one case, the experience consists of decades of expert experience with data structures, and in the other, the "basic-level distinctions" which help us in being completely confident of the identity and location of cats and mats in almost every realistic context. This latter kind of Cat/Dog/Animal bark/meow example is also very common in the OO literature. The moment the formalism is presented with a "more typical" example of a problem that doesn't have the benefit of having been formulated in advance to have essentially a single well-defined answer, the decomposition breaks down.

It should be a significant warning sign that "substitutability" has run into such fundamental problems when dealing what should be classically simple situations such as the Circle-Ellipse Problem which should be a trivial application of the theory to a ready made distinction taken directly from basic mathematics. As discussion of this problem quickly reveals, distinctions based on shared properties (or - shared descriptive state) can very easily give completely different answers to those based on shared function - and as Lakoff points out, there is no good reason why they shouldn't. It's interesting that many of the ways out of this conundrum point to the requirement for allowing an object to change "type" over its lifetime. Our system does not permit this, but the solution is related. We avoid such problems in two ways - firstly, by not making any type-based guarantees of function, and secondly - although we hold object instances to be immutable (minus *models*) from the very beginning point of their construction - because all referential power is mediated by the framework, we very easily allow one object to substitute for another in its architectural function by having the original object destroyed and a fresh one with a different "type" instantiated in its place. No observer is disturbed by the change since every observer is properly a pure function.

The avoidance of excess intention

What we need are schemes that facilitate the use of multiple interpretations of the same set of artifacts, and facilitate the process of switching from one such interpretation to another - not schemes that bake in exactly one, hierarchical and property-based interpretation into the very symbols we use to define and talk about the artifacts. We should avoid schemes that force the creator of an artifact to imbue it with what could be called ***excess intention*** - that is, extra constraints on the nature and activity of the artifact beyond those which the creator immediately requires. One classic example of such excess intention is *sequential intention* - imperative programming languages unnecessarily force the creator to commit to an exact sequence of executed instructions, which is usually far in excess of the real requirements underlying the goals he is interested in. However, another, more rarely identified kind of intention could be called *artifact boundary intention* - that is, a specific division of a problem space or a collection of state into a particular set of non-overlapping artifacts with definite boundaries. It is above all this kind of excess intention which the classical theories of *types* in all of their currently represented forms will force us into expressing - and it is the avoidance of this excess intention which lies above all behind our insistence that *Grades are Not Types*.

--

Footnotes

[1] Published in Philosophy and Social Hope, currently available on scribd at Ethics Without Principles

[2] described at The Structure of Scientific Revolutions, currently available at E.O. Smith Moodle

[3] Originally published in the ACM Transactions (Nov 94) as "A behavioral notion of subtyping", currently available at Ohio State

[4] From "The Complete Yes, Minister", pp 171-2 (Lynn and Jay, 1989) - "However, since there are virtually no goals or targets that can be achieved by a civil servant personally, his high IQ is usually devoted to the avoidance of error. Civil servants are posted to new jobs every three years or so. This is supposed to gain them all-round experience on the way to the top. In practice, it merely ensures that they can never have any personal interest in achieving the success of a policy: a policy of any complexity takes longer than three years to see through from start to finish, so a civil servant either has to leave it before its passage is completed or he arrives on the scene long after it started. This also means you can never pin the blame for failure on any individual: the man in charge at the end will say it was started wrong, and the man in charge at the beginning will say it was finished wrong .... Afterthought: considering that the avoidance of error is their main priority, it is surprising how many errors they make!"

[5] From "Boswell's Life of Johnson" (1784) "Sir, I have found you an argument; but I am not obliged to find you an understanding."

[6] See, for example, George Lakoff's "Women, Fire and Dangerous Things" (1987) or "Philosophy in the Flesh" (1999)