

Re: Question on LSP

Source: <http://coding.derkeiler.com/Archive/General/comp.object/2006-05/msg00239.html>

- *From:* "Dmitry A. Kazakov" <mailbox@xxxxxxxxxxxxxxxxxxxxx>
 - *Date:* Sun, 14 May 2006 16:25:35 +0200
-

On Sat, 13 May 2006 19:17:53 GMT, H. S. Lahman wrote:

Responding to Kazakov...

Why construction paradigms cannot be equivalent? In the sense that you were be able to impartially judge about their applicability in each concrete case?

Because construction paradigms have different goals. If all we wanted to do was write compact programs quickly and intuitively, we would all be doing functional programming. If our dominant problem is converting an RDB view to a UI view and vice versa, we would all be doing P/R. But OO development has a quite different goal: long term maintainability in the face of volatile requirements for large applications. Different goals require different construction practices.

Huh, it is like to say that different destinations require different construction of trains. In Europe it was definitely (and partially remains) so until all countries agreed on one width of the track. Obstacles are political, not technical.

Nice try but the analogy is at the wrong level of abstraction. To travel some distance over land one can fly, drive an automobile, or take a train. Each mode of transport satisfies different goals for the passenger in different contexts. Each mode requires substantially different transport mechanisms and those mechanism require quite

Re: Question on LSP

different interactions with the passenger.

They talk same language. And chairs by which they "interact" passengers are uncomfortable independently on destination. (:-))

you want an equivalence between three types there are many possibilities. Ones is:

[T]
A A
/\
V V
[T1] [T2]

Subtyping is transitive. So because T1 is a subtype of T it is also one of T2, because T2 is a supertype of T. And in reverse, T2 is a subtype of T1.

Picture me jumping up and down screaming, "No! NO! NO-NO-NO-NO-NO!"

Perhaps for purposes of designing some arcane language one can regard sibling subtypes as as somehow being subtypes of one another, but never in an OOPL context. T1 and T2 are semantically completely different types. They reflect /disjoint/ set membership in the problem space.

I don't propose to model disjoint concepts by equivalent types. I reserve equivalent types for *same* problem space concepts, which cannot be adequately modeled by one type. Typical example is numbers.

Peer-to-peer collaboration works for any problem and can be transformed to any implementation environment. Things like tripartite messaging don't. More to the point, broadcasting can always be emulated with multiple peer-to-peer messages so one doesn't need a special construct for those situations where it occurs.

But it is the same agrument RM people are using to justify limitations of their approach. You can prove 1+1=2 in 120 pages proof starting from ZF set axioms. The question is why youi should do that? Integer might be a set, but it would be awful to deal with integers in such representation. So the peer-to-peer collaboration might be complete (though I reserve doubts about it), but that does not make it KISS. I think double dispatch is quite natural and intuitive concept. If not then the whole idea of dispatch (and

Re: Question on LSP

so polymorphism) was rubbish.

But double dispatch is not readily implemented in all implementation environments. The OO paradigm needs to be implementable in all environments. Moreover, it must be implementable in an unambiguous manner so if one must bootstrap infrastructure to support double dispatch, one opens the can of worms for validating the infrastructure.

But same could be said about single dispatch and anything else. How OO paradigm could depend on some lame languages?

BTW, a litmus test for any theory, starting since Cantor times is – how do you construct numbers in your paradigm? (RM people are of course unable to answer this in any reasonable way.) What about your model? Let you have Z and Q. Along which relationship does "+" sent?

In a practical sense it is not relevant at the OOA/D level. Such things are really only of concern once hardware gets into the picture. So one needs to narrowly define semantic abstractions at the 3GL level for that sort of thing to be consistent with the hardware models. [Which involves compromises, like defining knowledge responsibilities in terms of memory data stores. That, in turn, leads to language pitfalls for access decoupling and the whole getter/setter debate.]

At any higher level of abstraction, one just accepts an existing general model for things like numbers and arithmetic operations. IOW, I don't need to "construct" them in my OOA/D; I already know what they are.

The song remains the same – how do you know that structures complex like numbers would never appear in the problem space? If you cannot handle numbers, why are you sure that you can anything else? (This is one reason why I count nGLs for lower-level abstraction than 3GLs.)

Re: Question on LSP

I am sure I can handle numbers because mathematics has provided workable rationalizations for the way the problem space works with numbers and it has provided hardware computing models so that software can work with numbers. The point is I don't need to reinvent that wheel to solve a problem in hand either in OOA/D or in a 3GL. IOW, I don't care how mathematics was able to rationalize numbers; I only care about manipulating them in a manner consistent with the problem space.

Yes. The question is not in reinventing numbers. It is in a possibility to do something similar when the problem space would require that. Specialized species become extinct, humans and cockroaches survive. (:--))

I didn't say a pointer isn't a type. I just said that its type semantics has nothing to do with the T type.

Hey, type's semantics is up to developer's discretion!

Sure, the developer must keep track of which type has been assigned to each pointer to manage complexity in creating the design, which is why the 'T' is in T*. But that has nothing to do with what a pointer type is. The pointer type is defined independently of T. IOW, the pointer type is the '*' part of T*.

But I don't define what it is. Subtyping is a relation which does not consider construction or the origin of types. It is merely an ability to get at the methods. Because (to me) there is no properties beyond methods, this immediately implies properties sharing.

But I am looking at it from the application developer view. In the context of OOPL semantics it is never polymorphic _once it is instantiated_. The syntactic sugar of T* is just evidence of the fact that a given pointer cannot be assigned to an object of any other type than the one the developer had in mind when instantiating it.

No, that's a different case.

1. The first one is polymorphic pointers. That is when a pointer points to a class. The target is of either type derived from T. The class is based on the set of types {T, T1, T2, S ...}. One have to distinguish a pointer to T

Re: Question on LSP

(and only T) and a pointer to the class rooted in T. They are of different types.

I don't think so. In my view you are talking about <at least> four different things here.

(1) The fact that a pointer type can be instantiated to any number of types when it is instantiated makes it — at best — a polymorphic type. But that does not mean that the semantics of an object reference is polymorphic.

But I don't want this. I want 1–1 mapping. So pointer is bound to a class type. This makes it polymorphic, but it still exactly two types in the relation.

(2) The semantics of Object Reference can be defined without regard to what the types are of the objects to which it is assigned. Those object types are completely orthogonal to the semantics of being an object reference.

I don't want this either. There is no untyped referential semantics.

(3) Whether T is subclassed does not matter to the semantics of a reference to a T type. There is only one T set regardless of how many subsets it may have. A pointer to T is a pointer to any object of that and only that set, which has nothing to do with whether T has been further subdivided in some other context.

Yes

(4) How an object reference may be instantiated is quite different than how an object reference is used or manipulated. Any type polymorphism only exists prior to instantiation. But once the object reference has been instantiated there is one and only one type associated with it throughout its life, regardless of how many unique objects may be assigned to it during that life.

But the type can be a class.

2. When methods are accessible via pointer. Your language could allow you to have polymorphic objects of the class based on the set of types {T, T*, T**, T***, ...}. On this object, of which you don't know, whether it is T

Re: Question on LSP

or T* or T** etc, a call to Foo will dynamically dispatch.

Methods aren't directly accessible via pointers in an OO context; only objects are. One of the severe problems with using type systems is that it encourages exactly that view because it marries message and method in the method signature.

[Yet another battle! (:–)]

That alone accounts for a large fraction of the really bad OOP code around; they are just FORTRAN and C programs with strong typing because that marriage allowed procedural developers to overlay procedural construction paradigms on the OOPs.

Consider an object whose behavior is described with a state machine. What is the type of the object? It can only be described in terms of the state action methods that it possesses. But the client sends an event message to the object. The object provides a mapping of the event identity and its current state to an action through the STT. Clearly the client does not access individual methods through pointers; it only accesses the object through a pointer to send it a message without even knowing what actions are available.

Sounds very untyped to me. Your messages are "public methods" to me. Your methods are "private methods."

The fact that procedural message passing has trashed the decoupling of message and method has — very unfortunately — been preserved in the 3GL type systems. One of several casualties is the idea that objects collaborate, not methods. In the OO paradigm a pointer can only point to an object, not to an individual method. Mapping to a method can only be done based upon the message identity (and current state for object state machines). IOW, one pointer; many messages and even more methods.

It is just duality of procedural and object views.

My claim is that formally and technically $T \leftarrow T_1$ is nothing better than $T \leftarrow T^*$. The case 1, is irrelevant to the issue, because it is covered by $T \leftarrow T^*$ (where "class T" is substituted for T).

I strongly disagree with this in an OO context because of (3) above. You are arguing that T^* is equivalent to $\{T_1^* \mid T_2^*\}$. That is quite true for polymorphic dispatch *IF* T is subclassed.

Re: Question on LSP

No, that's the case 1 you are talking about. It is the case 2, for which I claim that T* "subclasses" T, and the class has types {T, T*}. Further it also "superclasses" T, because "&" becomes a method of T.

It can be dynamically instantiated as in 2. And I disagree that time of bindings may influence semantics. It could be a sugar, but there is something you want to sweeten...

Where did I mention the time of binding? I just said that the language is able to hide the indirection because a pointer cannot be polymorphic once it is instantiated.

But it can! You mix types and classes which forces you to a very complex, yet limited types system.

Is reference an object? Does this object same that it points to? When you aggregate references, is the aggregate of referenced objects or target objects? Is the aggregate itself an object? Is it a reference? Do you need objects, references, aggregates of objects, aggregates of references, references to references, reference to aggregates of references of aggregates, and so on and so far all distinct entities, hard-wired in the language?

In order:

An object is not a reference. A reference is a 3GL implementation artifact that is independent of problem space entity abstractions (even if someone chooses to make it an object is in some language meta model).

So pointer object is not an object? This is what I meant. You have to postulate some things, which are not objects; have operations, which aren't methods; have types, which may not have classes; related to each other, but in some magic way. I don't want a language like this.

The reference is entirely separate and conceptually different than the target it points to.

The aggregate is of references. Is this a trick question? B-)

You don't allow aggregates of values?

Re: Question on LSP

Yes, but usually a computing space object. So the notion of 'object' may be somewhat flexible across 3GLs.

An aggregate is not a reference (though it may be accessed by references and it may aggregate only references).

So an aggregate containing a pointer to T is itself not a pointer. See, you have to introduce a third class of things: values, pointers, and aggregates of pointers. Where it will end?

However, I still don't see what this has to do with reflection. Or are you talking purely about language implementation mechanisms?

Reflection (in mathematics) is a principle which allows you to construct illegal things in a legal way. (:-)) You cannot have a set of all sets, but you can create something "like" this (using closures). You cannot have a set of types being itself a type, so you make a class type instead (and type tag refers to the specific type). You cannot have recursive types, so you use pointers. Reflection fundamentally involves some referential semantics to break infinite recursion.

--

Regards,
Dmitry A. Kazakov
<http://www.dmitry-kazakov.de>

.