

Re: Question on LSP

Source: <http://coding.derkeiler.com/Archive/General/comp.object/2006-05/msg00240.html>

- *From:* "H. S. Lahman" <h.lahman@xxxxxxxxxxx>
 - *Date:* Sun, 14 May 2006 16:56:32 GMT
-

Responding to Kazakov...

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.

But in an OO context T1 and T2 /are/ disjoint concepts by definition. That's what they cannot be subtypes of each other.

Peer-to-peer collaboration works for any

Re: Question on LSP

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 argument 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 you 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 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?

Single dispatch is always implementable at the 3GL level because all 3GLs employ procedural block structuring, message passing, and scope.

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

Re: Question on LSP

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.

Nonetheless types in general and subtypes in particular need to be defined with a consistent meta-semantics and there need to be rules about how they relate to one another. The type of '*' is independent of the type of 'T' and there can be no subtyping or substitution between those types.

Just out of curiosity, if you think of types solely in terms of methods, how do you define 'object'?

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 (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.

Re: Question on LSP

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.

It is only bound to a class type through instantiation, which was my point (4). Because that binding is 1:1, it eliminates any polymorphism that might have been possible at the type level.

(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.

There has to be that level of generalization. Otherwise one would have to define an infinite number of Object References in the language a priori to accommodate all possible applications. By separating the semantics of the Object Reference type from the target object type one has a much more compact and versatile mechanism because the association of types can be deferred to the binding (instantiation). But that only works if the type of Object Reference is completely independent of the target type.

(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.

A type /implements/ a class. There has to be an unambiguous mapping, but they are quite different things.

Re: Question on LSP

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 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 OOPL 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 OOPLs.

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."

Messages have identity and behavior responsibilities have identity. The mapping function (in this case the static states and transitions of the state machine represented in the STT) are rigorously defined. Those conspire to provide a very versatile description of problem space constraints on the sequencing of executing behaviors. That can be readily mapped into 3GL type systems.

But the reverse is not true. One cannot reconstruct the state machine purely from 3GL method calls without interpreting other things (event queue, STT, naming conventions, etc.). Thus the 3GL type system is actually less stringent about the constraints on collaboration.

The fact that procedural message passing has trashed the decoupling of message and method has — very unfortunately — been preserved in the

Re: Question on LSP

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.

OK, but that duality is crucial to constructing applications, which is one of the roots of our disagreement. One cannot think about constructing an OO application in terms of sequences of method invocations. But thinking in terms of sequences of operations is fundamental to both procedural and functional programming.

BTW, let's not lose sight of the original point in this subthread, which was your assertion that pointers reference methods. That is just not true in an OO context.

My claim is that formally and technically $T \leftarrow T1$ 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 $\{T1^* \mid T2^*\}$. That is quite true for polymorphic dispatch *IF* T is subclassed.

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

That does not make sense to me, even in a type system. The logical conclusion is that the language must define all the possible types. That is because subtyping is inherently static in nature so the language compiler/interpreter would have to know about all possible Ts ahead of time to deal with T^* subtypes. T^* only makes sense if one separates the T and the * parts and defers binding of them to <inherently dynamic> instantiation. But that binding is unrelated to subtyping.

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...

Re: Question on LSP

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.

Not in an OOPL. Once the pointer is bound to a type through instantiation you cannot reassign it to another type. That constraint is what allows the language to hide the indirection. Without that constraint one would also have to explicitly specify what type T^* is currently pointing to (S , U , V , etc.) in every invocation context. That would make the indirection an explicit two-stage affair. Instead of

- (1) get referenced identity (T^*)
- (2) access referenced target interface (T)

one would have

- (1) get referenced identity (T^*)
- (2) get actual type interface (S)
- (3) access referenced target interface (S)

The compiler could still hide the (3) indirection, but not the (2) step.

Not to mention the confusion it would create for the developer when T^* is actually pointing to an unrelated S type rather than the original T type in some context. B-) That sort of anarchy puts software developers in padded cells.

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

Re: Question on LSP

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.

No, a pointer is not an object in an OO sense. It does not abstract an identifiable problem space entity. It may be defined as an object in the language meta-model, but that is a quite different problem space. More important, if it is defined as an object in the language meta model, then that must be transparent to the OO application developer because the developer is relying on a much more restrictive definition of 'object'.

I don't see any magic:

- Objects abstract identifiable problem space entities.
- Objects have knowledge (attributes) and behavior (operations) properties.
- Objects are collected into sets (classes) based on property sets.
- Classes can be mapped 1:1 to types based on property sets.
- Objects are related to each other in a peer-to-peer manner.
- Object relationships are constrained by class-level static structure.
- Object collaborations navigate relationships by sending messages.

So long as the language supports that model, I couldn't care less how it decides to define its implementation mechanisms like pointers internally. In fact, I don't even care what implementation mechanisms the language employs. (I can easily transform an OOA/D model into straight C rather than an OOPL.)

Note, BTW, that there are no operations outside objects in an OO context. (Operation and method are synonymous in UML.) Also, all classes will have a corresponding type at the 3GL level and vice versa. (Though many of the types at the 3GL level implement classes from different spaces than the problem's space, notably pure computing space artifacts.)

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?

Actually no (except in private object implementations at OOP time). An object's knowledge responsibilities at the OOA/D level are always scalar ADTs. (The ADT, of course, can hide a complex aggregate of values.)

<aside>

One of the interesting things that the OO paradigm brings to the table is a flexible notion of logical indivisibility. In any given subsystem all the objects are abstracted at a consistent level of abstraction and their

Re: Question on LSP

collaborations reflect that. But other subsystems may abstract at a quite different level of abstraction. This is a very powerful alternative to functional decomposition for managing complexity.

I have worked on applications where a scalar attribute in a high level control subsystem actually expanded into more than a dozen classes with many instantiated objects and a total number of data values $\sim 10^{**8}$ in a lower level subsystem. That allows one to focus on high level control in the higher level subsystem without being distracted by a myriad of details. Conversely, it allows one to encapsulate details with much greater focus and control of scope. For large applications that is immensely important.

A side benefit is better change management. When the details are encapsulated at an appropriate low level of abstraction, requirements changes tend to be isolated to particular subsystems. In addition, the subsystem interactions tend to be much more stable over time because the subsystem interfaces can capture subsystem functionality at a higher level of abstraction.

</aside>

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?

Multiplicity in relationships is pretty fundamental to OO development (and the relational data model). There is a fundamental difference between 1 and many (functional and nonfunctional relationships in RDM terms). So aggregates in 1:* and *:.* relationships are <almost> always handled specially with collection objects at OOP time.

References are an essential part of OOPL syntax because they support identity by address in a memory model. Not to mention making things like aggregates much easier to manage.

Values are of interest in the context of knowledge responsibilities. One needs some mechanism for knowing something and 'value' is pretty much it. Since knowledge and behavior are quite different sorts of properties in an OO context and are handled quite differently (e.g., synchronous vs. asynchronous access models in OOA/D), it figures one needs to distinguish values from operations.

So there are several "classes of things" that are relevant to the OO paradigm. In fact, I would add objects, properties (subclassed to values and operations), and relationships to the list of fundamentally different sorts of things that coexist in an OO context. I could even make a case for message.

What's the problem with that? Sorting things out by flavor is a common technique for complexity management.

Re: Question on LSP

There is nothing wrong with me that could
not be cured by a capful of Drano.

H. S. Lahman

hsl@xxxxxxxxxxxxxxxxxxxx

Pathfinder Solutions -- Put MDA to Work

<http://www.pathfindermda.com>

blog: <http://pathfinderpeople.blogs.com/hslahman>

Pathfinder is hiring: http://www.pathfindermda.com/about_us/careers_pos3.php.

(888)OOA-PATH