JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2008**

Vol. 7, No. 4, May-June 2008

An enhanced form of dynamic untyped object-based inheritance

Anthony Savidis, ICS-FORTH, Greece

Abstract

We present an enhanced form of untyped object-based inheritance for classless languages, as implemented in our Delta language, comparing to the prevalent practices of delegation and embedding. Through a case scenario we reveal a design flaw of delegation that damages polymorphism and extensibility. Then, we show why embedding is impractical for object-based uninheritance (undoing inheritance on individual objects) and non-monotonic object evolution (dynamically adding or removing object members). We introduce dynamic object trees, adopting the metaphoric notions of inheritance from class-based languages, without compromising the compositional flexibility of untyped inheritance. We implement inherit and uninherit as library functions, discussing how our member lookup algorithm preserves monotonicity. Finally, we show that if prototypes are prototypical objects they may break their own invariant. To this end, we propose class objects as a more precise metaphor, implementing in the Delta language a function for dynamic mixin composition of class objects.

1 INTRODUCTION

The reported work concerns dynamic untyped object-based inheritance in the domain of classless languages. Such languages were quite popular in the past, putting forward the notion of prototypes [Lieberman86], [Smith94] as an untyped dynamic counterpart for classes, while they are less preferred today for the practicing of medium to large scale object-oriented software development. In contrast to the domain of classless languages, in which the reported work falls, dynamic class-based languages like Python and Ruby are very popular today, probably more popular than in the past. Usually this situation is attributed to both the lack of static type checking and to the unconventional inheritance programming models offered in comparison to class-based languages.

We show that the two prevalent practices for dynamic inheritance, being delegation and embedding, suffer from key shortcoming as recipes of dynamic inheritance – detailed summaries of delegation and embedding are found in [Cardelli96] and [Taivalsaari96]. In particular, we demonstrate why delegation harms polymorphism, information hiding and extensibility, and why embedding implementations become excessively complicated to support uninheritance and non-monotonic object evolution. We propose a new form of untyped object-based inheritance as implemented in a language named Delta [Savidis05], relying on object trees, showing how it overcomes the barriers of delegation and embedding. Finally, we discuss about a common misconception, i.e. that prototypes are prototypical objects, showing that prototypes should be objects of a separate class compared to the objects they generate during runtime. In this context, we propose the concept of class objects, providing a generic function to perform dynamic mixin composition on class objects, implemented in the Delta language.

We continue by setting the context of reported work, starting with untyped object-based inheritance, then providing a quick account of late binding in an object-based inheritance context. Finally, we discuss the case scenario that will be used to draw our arguments against delegation and embedding.

Elements of untyped object-based inheritance

In object-based languages the notion of a class is not mapped to an explicit language construct, but is used by convention as a design metaphor, with semantics borrowed from class-based languages. In this context, for classless languages, we adopt the following definitions:

- *object*: instantiation of the structure model of a class;
- *base object*: an object contributed from a base class;
- *subobject*: part of an object due to the members in the class-definition scope;
- *derived object*: any object that has base objects;
- *most-derived subobject*: the subobject of the most-derived class.

An example is provided in Figure 1 clarifying the use of these terms; on the leftmost part we illustrate the corresponding class-based structure. The way subobjects are internally glued together depends on how inheritance is implemented in a language. The dotted lines of Figure 1 only conceptually denote such implementation-dependent associations.

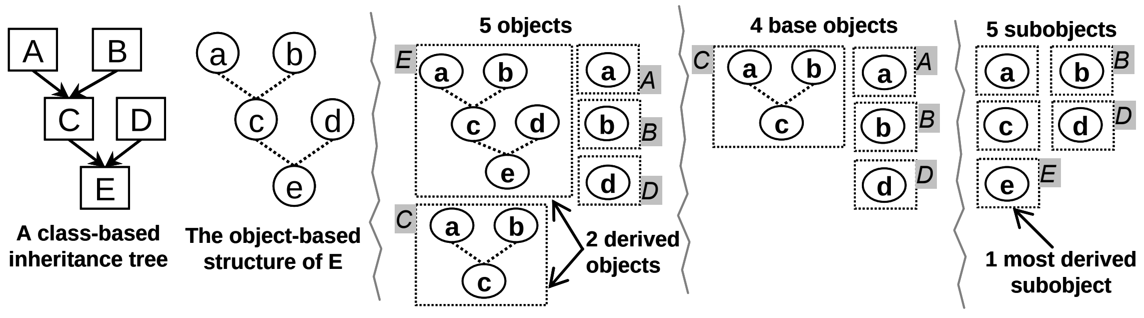


Figure 1: Sample class-based inheritance and its classless object structure (left part), with the various objects, base objects and subobjects (right part) – respective classes are shown in shaded rectangles.

In classless languages programmers still think in terms of class-based inheritance to design the corresponding object-based structures. Then, at the implementation, they should handle the assembly of subobjects via program statements, so as to recreate the designed object-based structures (as in Kevo [Taivalsaari93], Omega [Blascheck91], SELF [Smith95] and Lua [Ierusalimschy03]). For this purpose, classless languages offer varying linking or packing facilities, together with member lookup algorithms and suggested deployment patterns (e.g. traits and prototypes in SELF [Agesen00]). All the latter essentially constitute the inheritance framework of the language. This support of inheritance is radically different from class-based languages where the object-based structure (object model) is computed by the language (statically or dynamically) and the object assembly process upon instantiation is automated by the runtime system.

Generic 'inherit' and 'uninherit' operators Overall, classless languages differ mainly with respect to the object linking or packing semantics and the member lookup approach. The linking or packing of subobjects is supported as a runtime operation that we will informally call the *inherit* function, denoting its opposite as *uninherit*. Today, there are two dominant variations in implementing *inherit*: (i) delegation, linking of base objects via parental associations into acyclic directed graphs composed of subobjects; and (ii) embedding, packing base objects by substituting overridden methods with the addresses of the most recent versions. Usually, a delegation *inherit* means assigning a base object address to the parent slot of a derived subobject, while an embedding *inheri*t means concatenating or merging together a base object and a derived subobject. The vast majority of the existing classless languages implement *inherit* as variations of the delegation style. As we will show in our discussion, while implementing *uninherit* is straightforward via delegation, it turns to be severely complicated through embedding.

Late binding to most recent member versions

For the purposes of our argumentation we recall the notion of late binding in the perspective of untyped object-based inheritance. Late binding is the mechanism guaranteeing that in a polymorphic object the most recent version of any inquired member is resolved during runtime, independently of the referring subobject. Usually the referred subobject is mentioned as the callee or message recipient. Effectively, due to late binding, the following holds:

*For any object **x** and reference **y** to a subobject of **x**, all member inquiries via **y** dynamically bind to their most recent version within **x**.*

Normally, the most refined version may be located in a subobject other than the one through which the inquiry is made. A simple example is provided in Figure 2, showing an *Animated Button* object composed of three subobjects: while the calls to *Display* are made with different subobjects, all should bind to the most recent *Display* version residing at the *Animated Button* subobject. While this behavior of late binding is taken for granted in static or dynamic class-based languages, it is not entirely supported in delegation-based implementations of the *inherit* operator. We show that this deficiency is due to the parent-directed member lookup semantics of delegation.

Additionally, we demonstrate that in order to bypass this inadequacy one is practically led to an anti pattern damaging information hiding and extensibility. To prove these arguments we firstly define our subject case scenario. We put particular emphasis on this scenario, since it is crucial to convince that on the one hand it reflects common real-life programming needs, while on the other hand the existing techniques fail to support it.

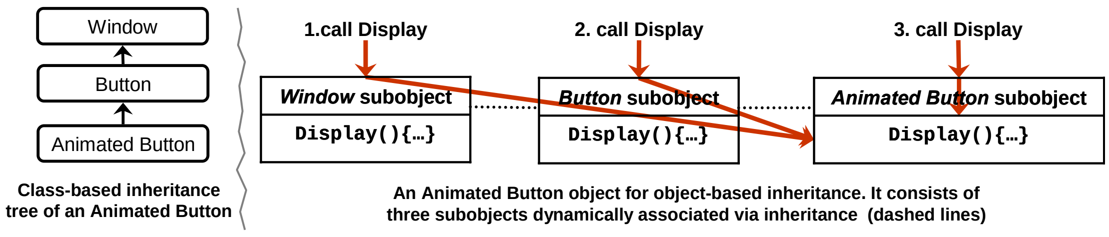

Window
Button
Animated Button
Class-based inheritance tree of an Animated Button
1.call Display
2. call Display
3. call Display
Window subobject
Display(){…}
Button subobject
Display(){…}
Animated Button subobject
Display(){…}
An Animated Button object for object-based inheritance. It consists of three subobjects dynamically associated via inheritance (dashed lines)

Figure 2: An Animated Button object in the object-based world (right part) for the class-based scheme on the left; calls to *Display* with different subobjects bind to the most refined *Display* version.

2 SUBJECT CASE SCENARIO

Part I Consider the general case where for a base class, the address of every created object needs to be stored in some kind of a holder object. The latter uses such stored references to invoke specific base methods during runtime. In particular, let's assume a *Window* base class, a *WindowManager* holder object, and a *Display* method for *Window* objects. The latter concerns typical windowing systems, offering rich collections of derived classes called widgets (e.g. buttons, toolbars, text fields, scrollbars, etc.), while letting programmers introduce further specialized behaviors such as auditory or animated buttons. The general scenario is very common, reappearing in numerous situations: symbols / symbol table (compiler), graphic elements / display manager (graphic editor), game characters / AI manager (game engine), agents / coordinator (agent systems), etc.

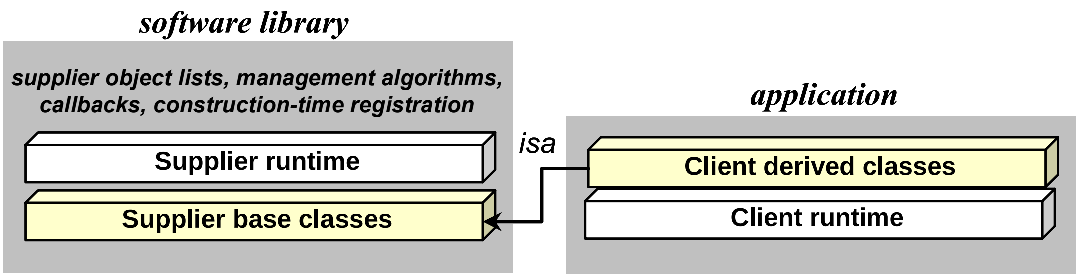


Figure 3: Part I of our scenario is common in software libraries offering supplier classes (to be extended by clients) with comprehensive runtime management mechanisms (like window managers or game engines).

As illustrated in Figure 3, this scenario is typical to software libraries combining supplier classes and runtime systems that automatically manage all created supplier instances (like the case of windowing systems).Following our scenario, window managers keep lists of all created windows for processing purposes. In all known systems such bookkeeping is handled upon construction and destruction time through calls placed in the Window base-class constructors and destructor. Since references to Window objects are stored in the internal lists, the window manager may invoke Display only via such Window references.

Next, we implement such a *Window* class in a classless language context via a prototype. Here we adopt the common practice: a prototype is a prototypical object, while to avoid exposing a reference variable we provide a function returning the reference of the prototypical object. In our case, we define a function named *Window*, meaning *Window()* returns the address of the respective prototypical object. A *Window* prototype encompasses a constructor method - we name it *New* – and various data members with default values. Construction via the *New* method is carried out by replication (copy) of the prototypical object, implying the automatic registration to the window manager lists. In Figure 4 we sketch the *Window* prototype implemented in the Delta language and the *Display()* method of the *WindowManager* object in pseudo code (one window manager object exists during runtime, i.e. the analogy of a singleton class).

```
function Window() {
  static proto;  ←Reference to prototypical object hidden as a local static variable
  if (isundefined(proto))  ←Prototype object is initialized only due to first call
         proto = [
           {.Display      : (method(){ … })},
           {.Handle       : (method (event) { … })},
           {.New          : (method(){  ←User-defined constructor method
'objcopy' lib func performs  shallow copy→   local win = objcopy(self);
                                             WindowManager.Register(win);
                                             return win;
                                       })},
                ... Rest of Window data members (with default values) and methods...
         ];
  return proto;  ←The prototype is returned as a factory to make objects via New()
}
```

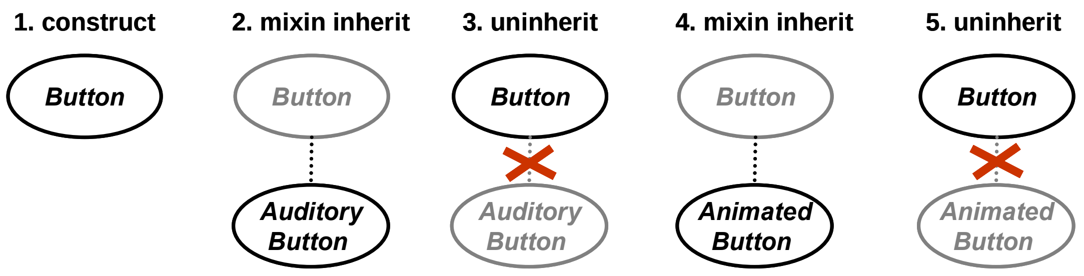
*WindowManager.Display() { for each window **w** do **w.Display**(); }*

Figure 4: (a) The *Window* prototype in the Delta language with the *New* constructor performing copy of the prototype and registration in *WindowManager* singleton (b) The *Display()* method of *WindowManager*.

We briefly explain a few of the syntactic and semantic details of Delta. Objects are created by the evaluation of an *[...]* expression, with initial members enumerated inside the square brackets as *{ key : expr }*, key restricted to strings and numbers, where *.key* is syntactic sugar for *"key"*. For example, `{.Display : (method(){ … })}` introduces a member with key *"Display"* and value the supplied implemented method. Given an object *a*, *a.x* is syntactic sugar for *a["x"]*. Finally, *objcopy(x)* performs a shallow copy of the subobject referred by *x*.

Part II The second part of our scenario concerns attaching (detaching) a derived object on (from) a base object during runtime. Technically, the latter implies the support for inherit and uninherit as well. Focusing on inheritance only, in the class-based world the scenario reflects mixin inheritance with generic derived classes. In particular, let's assume a *Button* object that during runtime inherits to an *Auditory Button* object (see Figure 5, steps 1 and 2). Apparently, the original *Button* will hereafter behave as an *Auditory Button*. Latter, this link is undone, meaning we have a normal *Button* again (step 3). We similarly reapply mixin inheritance over the same *Button* object turning it to an *Animated Button* (step 4) and back to a normal *Button* (step 5). During runtime, in-between such actions, the window manager will normally display all windows by calling its *Display* method, as illustrated under Figure 5 (bottom-right part, calls to *WM.Display*).

Our general scenario is related to all cases where a behavior needs to be dynamically added or removed. Its specialization is based on the example of [Bracha92], originally with a *Window* base class and a *Bordered* mixin class, adapted as follows for the object-based world: (i) a *Window* object may be controlled by the user to have a border or not, hence *Bordered* mixin is applied dynamically on distinct instances; and (ii) we use *Button* in place of *Window*, and *Auditory* / *Animated Button* in place of *Bordered*.



```
1: button = Button().New();                    … WM.Display();
2: auditory = AuditoryButton().New(button);    … WM.Display();
3: uninherit(auditory, button);                … WM.Display();
4: aninated = AnimatedButton().New(button);    … WM.Display();
5: uninherit(animated, button);                … WM.Display();
```

Figure 5: For the Button object of step *(1)* we apply a mixin *Auditory Button* at step *(2)* which is later un-inherited at step *(3)*, applying another mixin *Animated Button* at step *(4)*, finally un-inheriting back to a *Button* at step *(5)*; *WM* is an abbreviation for the *WindowManager* object.

The need for mixin inheritance on individual objects has been already identified in class-based languages, like in the Decorator design pattern [Gamma95] prescribing the dynamic installation of derived behaviors (decorations) on distinct objects. The implementation of the *AnimatedButton* prototype in the Delta language is outlined in Figure 6; *AudidoryButton* prototype is similar.

```
function AnimatedButton() { ←Mixin proto over Button, derived from Window
   static proto;
   if (isundefined(proto))
         proto = [
               {.Display: (method(){ … })}, ←Refines Button.Display
               The 'button' argument below is the object on which the mixin is applied
               {.New      : (method(button) {
                                 local anim = objcopy(self);
                                 inherit(anim, button);
                                 return anim;
                           })}
               ... Rest AnimatedButton data members (with default values) and methods...
         ];
   return proto; ←Prototype returned as with Button class
}
```

Figure 6: The *Animated Button* mixin prototype over *Button* in the Delta language; the implementation of *Auditory Button* is similar and is skipped for clarity.

What these scenarios essentially test The first part of our scenario tests late binding. The window manager stores references of *Window* subobjects, invoking their *Display* method. Each *Window* subobject may be part of a derived object, like a *Button* or *Menu*, further refining *Display*. Hence, while the window manager uses references to *Window* subobjects for *Display* invocations, resolution to the most recent *Display* method should be guaranteed. We will show that a delegation-based inherit fails in this respect.

The second part of our scenario tests the support for object-based uninheritance. In general, object-based mixin inheritance or uninheritance is required in all cases that adding or removing mixins during runtime makes sense. In this context, the Decorator design pattern demonstrates that such a need does emerge in real practice. Apart of uninheritance, implying 'cutting' inheritance links on-the-fly, we additionally require the support to dynamically add or remove members on individual subobjects. In class-based languages, if the latter is supported on individual classes, it leads to non-monotonic class evolution [Kniesel00]. In untyped object-based inheritance where inheritance is not subtyping the analogy is dynamic object evolution. We will see that embedding-based implementations for *uninherit* and dynamic object evolution are excessively complicated.

3 RELATED WORK

Currently, there are numerous languages supporting untyped object-based inheritance, relying on variations of either the delegation or embedding style. The differences among languages adopting the same style concern various enhancements, like lookup extensions, introspection support, multi-threading capabilities, dynamic compilation, built-in support for prototypes, etc., just to name a few. Despite the existing variations, all different implementations comply with the following: (a) delegation is built around dynamically-managed parental associations that are taken during runtime to recursively resolve invocations of methods within parent objects; and (b) embedding applies a non-reversible concatenation of subobjects, changing the addresses of refined methods in base objects to reflect the most recent versions, while also saving the previous addresses.

We consider SELF [Smith95] and Kevo [Taivalsaari93] to be the most representative cases of object-based languages regarding delegation and embedding, with the exception that Kevo is also a statically-typed language. SELF supports assignment of values to parent slots, so that delegation graphs can be constructed, offering alternative versions of the basic lookup algorithm supporting detection of cyclic delegation paths. A more recent untyped language is Lua [Ierusalimschy03]. In Lua parent objects are called 'meta-tables', following the Lua approach where objects are associative tables.

Apart of implemented languages, δ [Anderson02] is an imperative typed object-based calculus supporting inheritance by delegate editing. It enables non-monotonic object evolution by supporting a built-in operation for adding or removing methods. Clearly, since the δ calculus focuses on typed inheritance, emphasizing type safety and type inference, it is not related to the reported work. However, we refer to δ for two reasons. Firstly, because delegation in the δ calculus still relies on traversal over parental links, as denoted by the semantics of the Look′ operation of the δ calculus, meaning typed delegation inheritance reflects similar lookup mechanics to untyped inheritance. Secondly, to clarify that our Delta language - acronym for 'Dynamic embeddable language for extending applications' - is not related to the δ calculus, since they happen to be pronounced the same.

Embedding was introduced in the context of the Kevo typed language [Taivalsaari93] as an alternative to delegation, proposing inheritance via copying and composition. Currently, all known embedding languages are typed. Obliq [Cardelli95] is an embedding language designed to primarily advance the support for programming of distributed computations rather than embedding inheritance itself. Besides Kevo, Obliq and Omega [Blascheck91], no other embedding-oriented languages are known.

4 SHORTCOMINGS OF DELEGATION

As shown in Figure 7 (left part – adapted from [Kniesel00]) lookup follows parental links: once an owner of the referred method is found (step 3), the method is invoked binding self to the original method recipient subobject (step 4). The lookup is always initiated at the recipient subobject (step 1), applied recursively upwards (step 2); some languages support alternative termination conditions, besides the default that is to stop once they match the inquired method name.

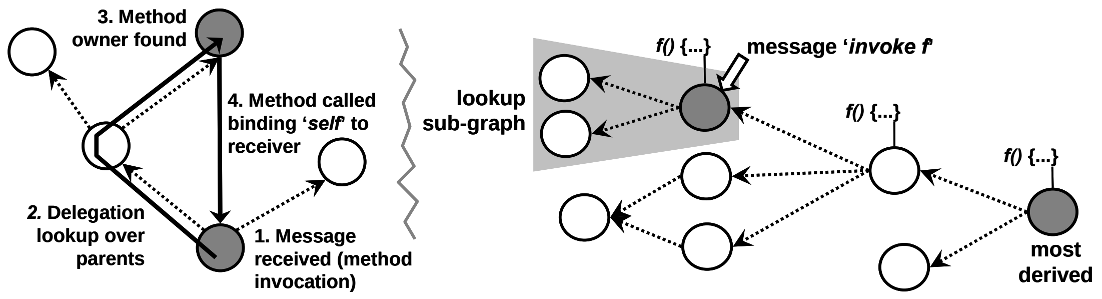

3. Method owner found
4. Method called binding 'self' to receiver
2. Delegation lookup over parents
1. Message received (method invocation)
lookup sub-graph
f() {...}
message 'invoke f'
f() {...}
f() {...}
most derived

Figure 7: Delegation-based binding (left), restricted lookup if not using the most-derived subobject (right).

Following Figure 7 (right part), let's consider method *f* refined by the most-derived subobject, invoked using another subobject of the delegation tree. Since delegation performs lookup over parental links, the search space is narrowed down to the sub-graph reachable from the message recipient (method invocation) subobject (see Figure 7, shaded sub-tree). Clearly, the most recent version of *f* is not locatable in the restricted lookup space, meaning late binding fails to resolve to the most refined method version.

Now we map the example of Figure 7 to our specific scenario: in place of *f* we have the *Display* method, invoked through a *Window* subobject, whose address is stored in a *WindowManager* list, while the most-derived subobject is an *AnimatedButton* or *AuditoryButton* refining *Display*. Then, in a way similar to Figure 7, none of *x.Display()* calls performed in *WindowManager.Display*, *x* being a *Window* subobject, binds to the *Display* refined by the *AnimatedButton* or *AuditoryButton* subobjects. The reason is that, since delegation-lookup follows parental links (i.e. base objects) a lookup initiated at a *Window* subobject never reaches a derived *AnimatedButton* or *AuditoryButton* subobject. In conclusion, delegation-based *inherit* fails to support late binding in our scenario.

Trying a covariant self The support of covariant *self* is discussed in [Abadi96] in a typed-inheritance context, proposing to explicitly qualify as *Self* the return type of methods that aim to return *self*. Such a language feature can be used to resolve the delegation problem, but only for class-based languages, as follows: adopting the notation of [Abadi96], we introduce *method most_derived(): Self is self* in a common super-class, which due to *self* covariance, guarantees to return the most derived subobject independently of the caller. Then, in the *WM.Display*, we change the style of the call to *x.most_derived().Display()*. This ensures method invocations are always made with the most-derived subobject, so the call binds by definition to the most recent *Display* version.

Since the previous wrap-around is applicable only to class-based languages we investigate the possibility to implement it in an untyped delegation context. Technically, the covariant *self* is a late binding mechanism for the *self* construct. The only way of emulation in untyped inheritance is via a method *most_derived() { return self; }*, adopting the same modified call style as before. However, *x.most_derived().Display()* is equivalent to *x.Display()* for the following reason: The call to *x.most_derived()* by the *WindowManager*, having *x* a first message recipient, implies *most_derived()* is invoked binding *self* to *x*, thus returning *x*. In other words, this implementation of *most_derived()* suffers from the same problem, that is restricting lookup to the sub-graph reachable by the message recipient, failing to return the most-derived subobject.

Overall, there is no appropriate wrap-around in untyped delegation to resolve the issue of lookup dependency on the first message recipient. So, as a last option, we investigate the possibility to impose as a programming discipline that all method invocations are made only with the most-derived subobject in a delegation graph, i.e. forcing a most-derived first message recipient rule.

Imposing most-derived first-message recipients Through this rule we force method invocations to be made only with the most-derived subobjects. This way, the delegation lookup space is by definition the entire graph. So, it is guaranteed that the most-recent versions of called methods are always resolved. While the rule seems to sort things out, we show that it harms extensibility, information hiding and abstraction.

To comply with this rule we should ensure that the *Window Manager* stores references to most-derived subobjects – e.g. *AnimatedButton* subobjects. But the latter requires refactoring of the implementation to transfer *Register* calls outside the *Window* constructor, at every point that *AnimatedButton* or *AuditoryButton* objects are created. Apparently, we may not hook such calls inside the *AnimatedButton* or *AuditoryButton* constructors as we will be faced with the same issue if we similarly derive from such objects. Practically, the above modifications, besides the apparent code replication, introduce forward dependencies on inheritance: every time object-based inheritance is applied, the original source code must be modified. For instance, if we aim to add / remove a dynamic mixin to an object, like applying an *AnimatedButton* subobject over a *Button* object, we should substitute in all registries the base object with its mixin subobject, and vice versa. Additionally, the rule imposes to reveal intrinsic APIs, such as the *WindowManager* object, not necessarily designed to be available to client programmers. This damages information hiding, reducing extensibility and modifiability of the underlying window manager system. Overall, the rule itself is evidently problematic, since, besides extensibility and information hiding that are damaged, abstraction and polymorphism are broken.

Closing remarks Our conclusion may seem surprising, not only because many delegation-based classless languages exist, but also due to the proof of equivalence between delegation and class-based inheritance reported in [Stein87]. However, in the latter it is assumed that member inquiries are always resolved in the entire delegation graph, something that for untyped delegation implies a first-message recipient rule. The reason that this problem of delegation was not spotted in the past is attributed to the role of classless languages in the course of real practice: as discussed in [Ousterhout98], dynamic languages were adopted for scripting / gluing purposes, rather than for building or extending comprehensive software libraries. The latter was primarily done via static class-based languages. Our scenario and argumentation is based on the assumption that library software may be implemented via classless languages as well.

5 SHORTCOMINGS OF EMBEDDING

Embedding languages allow programmers to perform unrestricted composition of objects from subobjects, internally automating the substitution of method addresses of base subobjects by the addresses of refined method versions from derived subobjects. In an untyped context, the notion of derived or base is defined by convention, not by type semantics, considering constituents to play the role of base objects and the composition outcome to be a derived object. Due to method address substitution upon composition, method invocations are guaranteed to take place with the most recent address. As mentioned before, all known embedding languages are typed.

The major drawback of embedding is the severe implementation complexity in supporting two key features: (i) uninheritance, following our scenario with dynamic mixins, supporting also cancellation of inheritance links independently of how they are set – e.g., one could cut multiple chained mixins altogether with a single *uninherit* call; (ii) dynamic removal or addition of members on individual subobjects. In particular, following the compositional style of embedding, uninheritance requires object disassembly support. Assuming that disassembly is applied on the inverse ordering of composition, in every subobject a stack may be kept storing the addresses of the previous method versions before composition is applied. Then, 'un-embedding' may restore the previous method addresses simply popping from the stack. Such an implementation is simple, but it works only for undo-like uninheritance. Now, let's alter the restriction that dynamic uninheritance occurs only in the inverse order of composition. Then, no backup / restore technique will suffice. Additionally, once we allow individual methods to be added or removed, things get even more complicated. Overall, the focus on method address substitution on concatenation, rather than on storing explicit inheritance links, does not allow to compute the most-recent method addresses of methods when uninheritance in freely applied. It comes by no surprise that we entirely miss models or implementations of uninheritance via embedding.

6 DYNAMIC SUBOBJECT TREES

Definitions

Our inheritance model relies on dynamic associations of the form $\alpha \swarrow \beta$, with the metaphoric interpretation $\boldsymbol{\alpha}$ *derived from* $\boldsymbol{\beta}$ and $\boldsymbol{\beta}$ *inherits to* $\boldsymbol{\alpha}$. The symbolism $a \leftarrow \beta$ denotes that $\boldsymbol{\beta}$ *is a base object of* $\boldsymbol{\alpha}$, representing direct or indirect derivation of α from β. The establishment of an inheritance association $\alpha \swarrow \beta$ is permitted if and only if the following precondition holds:

$$\alpha \neq \beta \ \wedge \neg\, \beta \leftarrow \alpha \ \ \wedge \ \neg\, (\exists \gamma :\ \gamma \neq \beta \wedge \gamma \swarrow \beta)$$

We adopt the metaphors of class-based languages, like 'inherits', 'derived' and 'base', rather than the ones of delegation languages, i.e. 'delegation', 'delegated' and 'delegator'. The three conjunctions formalize the fact that an object: (a) cannot inherit from itself, i.e. no trivial cycles; (b) cannot inherit from any of its directly or indirectly derived objects, i.e. no cycles; and (c) can inherit to at most one object, i.e. the relationship results into trees. From the three rules, the third one, restricting to a single descendant object, may seem rather unnatural. We explain later that this rule puts no restrictions on inheritance, while simplifying the internal implementation and the external programming model.

Inheritance control functions

The management of inheritance object trees is facilitated through a compact set of library functions. Instead of a procedural API, a parent-slot editing model could be provided, as in the SELF language. In our case we decided to adopt an API style, since we consider the slot model to be more close to the underlying implementation. The following functions are provided for controlling inheritance associations, *α* and *β* being subobjects:

- ***inherit***(*α*, *β*): Introduces *α*↙*β* adding *β* as the *leftmost* parent of *α*.
- ***uninherit***(*α*, *β*): It cancels the inheritance association *α*↙*β*.
- ***isderived***(*α*, *β*): Returns whether *α*←*β*.

Since inherit(α,β) inserts β as the leftmost parent of α, inherit is not cumulative, i.e. inherit(α,β) inherit(α,γ) ≠ inherit(α,γ) inherit(α,β). This property is made explicit in the programming model, since, as we discuss next, the tree structure affects the outcome of the lookup process. The implementation of inherit and uninherit in the Delta language, together with the basic data-structure for subobjects, are provided in Figure 8; the simplicity of the implementation is apparent.

```
SubObject {
    Members:       dictionary                    ← Hash table with all members (identifier – value)
    MostDerived:   SubObject reference           ← The most-derived subobject of the tree
    Derived:       SubObject reference           ← The single descendant (derived) subobject
    Bases:         list of SubObject reference   ← Base sibling subobjects ordered left-to-right
    MyTree:        list of SubObject reference   ← The tree subobjects ordered breadth-first left-to-right;
}                                                   this list is shared by all subobjects of the same tree.
```

```
inherit (a, b) {             establish a↙b       uninherit (a, b) {          cancel a↙b
    clear a.MyTree and b.MyTree                      clear a.MyTree and b.MyTree
    insert b in front of  a.Bases                    remove b from a.Bases
    b.Derived = a                                    b.Derived = nil
    t = BFS left-to-right from a.MostDerived         b.MostDerived = b
    _link(t, a.MostDerived)                          tb = BFS left-to-right from b.MostDerived
}                                                    _link(tb, b.MostDerived)
_link (t, c) {                                       ta = BFS left-to-right from a.MostDerived
    for each x in t do                               _link(ta, a.MostDerived)
        x.MostDerived = c, x.MyTree = t          }
}
```

Figure 8: The inherit / uninherit functions as implemented in the Delta language, together with the data-structure for subobjects; MyTree is kept up-to-date to store the addresses of tree subobjects ordered from the most-derived to the least-derived.

Member lookup

Following our discussion, to resolve the most-recent version of a member in an inheritance tree it suffices to locate the first subobject owning the member by examining subobjects in a breadth-first left-to-right search, starting from the root (most derived). Since for every subobject *x*, the list *x.MyTree* holds the subobjects ordered this way, we need only scan this list sequentially and return the first found subobject that owns the inquired member. The latter is reflected in the lookup algorithm of Figure 9 (left part).

```
lookup (o, m) {
    for each x in o.MyTree do
        if  m is found in x  then
            return x.m
    return nil
}
```

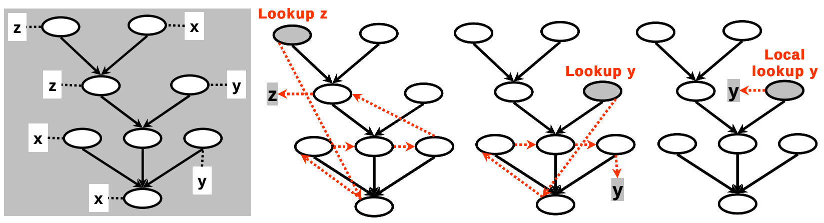


Figure 9: The simple lookup algorithm of the Delta language with a few examples; local lookup concerns member inquiries that are qualified to be resolved only in the caller subobject context.

The lookup algorithm is an untyped counterpart of member resolution ordering, the latter produced in class-based languages by preprocessing the hierarchy structure so as to identify a monotonic class linearization sequence. In Figure 9 (right part), a few examples are provided illustrating the alternative search paths to resolve particular members within an object inheritance tree. From the programmer's point of view, the member binding method is very easy to follow, being compliant with the behavior of late-binding in class-based languages and the adopted metaphoric notion of inheritance associations in object trees (i.e. what 'base', 'derived' and 'most derived' essentially imply). Next, we discuss two key properties resulting from the member lookup algorithm, namely lookup monotonicity and subobject substitutability. Then, we continue by justifying the rule restricting base objects to have a single descendant subobject.

Lookup monotonicity Lookup monotonicity concerns only class-based inheritance, requiring member resolution within base and derived classes following the inheritance-specific class ordering. In our model, although subobject associations represent metaphorically the notions of inheritance, those are directly reflected and preserved by the member lookup algorithm in a way similar to class-based languages. Although inheritance is untyped, once the notions of inheritance are explicit in the programming model, it is critical to prove that lookup monotonicity is guaranteed. In this context, it is trivial to prove that the breadth-first left-to-right ordering, for any given subobject tree, is monotonic, i.e. order preserving, assuming the ordering relationship defined below:

$$\boldsymbol{\alpha} < \boldsymbol{\beta} \; : \; \boldsymbol{\alpha} \swarrow \boldsymbol{\beta} \vee \boldsymbol{\alpha} \textit{ left sibling of } \boldsymbol{\beta}$$

Following the previous definition of inheritance associations, the two disjunctive conditions have the metaphoric interpretation ***α*** *derived from* ***β*** and ***α*** *most recent sibling of* ***β***, being equivalent to $\boldsymbol{\alpha} < \boldsymbol{\beta}$: ***α*** *most recent than* ***β***. Consequently, the member lookup sequence is monotonic, since it preserves by definition the inheritance-oriented ordering, visiting subobjects sequentially, from the most recent to the least recent.

Subobject substitutability This is a very important property of the member lookup algorithm, regarding all late-bound member inquiries. In particular, given any T tree of distinct subobjects $\alpha_1,\ldots,\alpha_N$, any expression E involving subobject $\alpha_j \in T$, and the notation $E\{\alpha_j{:}\alpha_\kappa\}$ implying substitution in E of every α_j by α_κ, the following holds:

$$\begin{array}{llll} \mathit{eval}\ E & = \mathit{eval}\ E\{\alpha_j{:}\alpha_1\} & = & \mathit{eval}\ E\{\alpha_j{:}\alpha_2\} = \ldots \\ \mathit{eval}\ E\{\alpha_j{:}\alpha_{j-1}\} & = \mathit{eval}\ E\{\alpha_j{:}\alpha_{j+1}\} & = \ldots & \mathit{eval}\ E\{\alpha_j{:}\alpha_N\} \end{array}$$

In other words, assuming late-bound member inquiries, the subobjects of the same tree are referentially equivalent to each other, as we can substitute any subobject with another in a program expression and still gain the same evaluation result. The latter is a more strict form of LSP [Liskov87]. More specifically, LSP introduces substitutability among objects of related derived classes as a desirable class-design property to promote polymorphic functions, requiring reasonable behavior of a polymorphic function after substitution. In our model, the subobjects of the same tree are conceptually related in the same way as the distinct subobjects of a polymorphic object in class-based inheritance, meaning they conceptually map to a family of related classes. Hence, substitutability should apply. Moreover, due to the previous property, after substitution a polymorphic function is guaranteed to have the same behavior. From the discussion on the shortcomings of delegation it is clear that this type of referential equivalence among subobjects of the same object graph is not supported in delegation, since the behavior may vary depending on the first-message recipient for successive method invocations. We consider that this is technically the 'Achilles heel' of delegation, regarding support for untyped object-based inheritance.

Why a single descendant rule

The rule forcing base objects to donate to at most one derived subobject is of key role in our method as it turns the inheritance graph to a tree. It should be noted that this rule applies only to objects in untyped inheritance, meaning it is irrelevant to class-based languages. This clarifying remark is necessary to prevent interpret the rule in a class-based context, as restricting base classes to at most one derived class is out of discussion. To prove that a single descendant is not restrictive in our inheritance model, we review the possible positions of a subobject in an inheritance graph without this rule. In any such graph, for a given subobject x one of the following may hold (see Figure 10 left part):

1. x donates to a single subobject (cases O_0, O_2, O_3)
2. x donates to none, so it is the most-derived subobject (case O_4)
3. x donates to more than one subobjects (cases O_1)

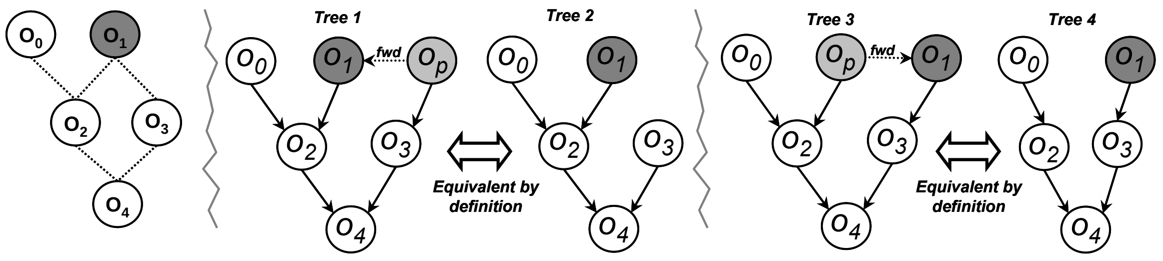


Figure 10: Sample subobject graph (left), and the two equivalent 'by definition' object trees (right), due to our lookup algorithm.

We will now transform the subobject graph of Figure 10 to a tree having equivalent member resolution behavior in our inheritance model. Following Figure 10 – see Tree 1, since O_1 may donate only to a single subobject, we choose to retain $O_2 \Swarrow O_1$ while introducing O_p and $O_3 \Swarrow O_p$, with O_p a proxy object merely forwarding all member inquiries directly to O_1 (dashed arrow). Due to the presence of O_p all inquiries not resolved in O_3 are eventually resolved in O_1 as if O_1 was also donating to O_3. Apparently, although the final tree is equivalent to the initial graph, it is impractical to require programmers explicitly introduce such a proxy. We show that the latter is never required.

The removal of O_p results in the *Tree 2* of Figure 10. Then, due to the lookup algorithm, $\forall\, x \notin \{O_0, O_2, O_3, O_4\} \wedge x \in O_1 \Rightarrow O_3.x \equiv O_1.x$, i.e. inquiries to O_3 for members found only in O_1 are automatically resolved in O_1. Since the proxy is redundant, the *Tree 2* of Figure 10 is equivalent to the initial graph. If we alternatively choose to retain $O_3 \Swarrow O_1$ we gain another equivalent tree, *Tree 4* of Figure 10. Hence, for the example graph, programmers have two plausible alternatives, linking the donor object O_1 to either O_2 or O_3. The reason that the single descendant rule puts no restrictions in our model is that the lookup algorithm guarantees to resolve inquiries to subobjects for members actually owned by other subobjects of the same tree. Consequently, there is no need to link multiple subobjects of the same tree to a single base once the latter is already linked to any of these subobjects.

7 FROM PROTOTYPES TO CLASS OBJECTS

Prototypes should not be prototypical objects

Prototypes are used to emulate classes in untyped object-based languages [Leiberman86], being normal prototypical objects differentiated only due to their distinctive design role: to produce other objects via replication. We will show that once prototypes are designed to comply with Design by Contract [Meyer97] their traditional treatment as prototypical objects turns their use to totally impractical.

Prototypes should exist during runtime prior to the production of any respective object. Intuitively, this is an exceptional privilege that may cause prototypes to possess state that is not plausible for the objects produced from it. While this issue was identified very early, the incorrect deployment of prototypes as normal objects was treated as a cause of failure that programmers had to simply avoid [Smith94], not as a fundamental design flaw. Theoretically, turning prototypes to prototypical objects implies that prototypes are of the same class as the objects they produce. Then, putting also Design by Contract into the game, the following should hold: *prototypes satisfy the class invariant of the produced objects*. We demonstrate that this rule makes the use of prototypes impractical; we build our argument incrementally:

- If there is a single object of a class *C* then this should be the prototype.
- Therefore, a prototype *C* cannot be used when there is no *C* object.
- Therefore, to use a set of prototypes $C_1,..,Cn$ there should be at least one object per C_i prototype.
- Therefore, inheritance can be applied on prototypes $C_1,..,Cn$ only when at least one object per C_i prototype.
- Therefore, inheritance is applicable at runtime only when at least one object per involved prototype exists.

However, to keep an object of every class alive during the whole execution lifetime is generally wrong: the creation of objects depends on the application semantics defining when and why respective objects should be available. An object of a class cannot exist unless the application state reaches a point implying that it should come to construction. In this context, we assume that the invariant of every supplier class includes the following condition: the object exists as a result of correct application execution. Such a rule constitutes a generic global criterion for object correctness that can be easily asserted (e.g., using application-level object bookkeeping) to be included in class invariants. This rule implies that given any object satisfying the class invariant, an exact copy is not directly guaranteed to also satisfy the invariant. The reason is that object copying and generation should be governed by the application semantics. Consequently, to unconditionally force that for all classes at least one fully functional object should be retained during execution may contradict to the application design, and inherently to the respective class invariant. Following the previous discussion, instead of prototypes being technically a wrong metaphor, we propose the term class objects, capturing more precisely their design role. Through this differentiation, in the design domain, we now distinguish two different classes:

- The class concerning the produced objects, say *A*
- The class concerning the class object itself, say *A_IMPL*

The previous differentiation makes sense only in an object-based world where classes are implemented as objects. In this context, our remark reveals that the behavior of a class object is different from the behavior of the produced objects, meaning they semantically map to different classes. Moreover, while the A class never exists as an explicit object, i.e. it is only an artifact of the design domain, it is explicitly reflected in the contractual obligations of objects produced by A_IMPL objects. More specifically, when an A_IMPL class object creates A objects during runtime, it has to install on them all Design by Contract members implied by class A. Finally, reflecting our remark, given any class A and object O the following holds:

***O** produces objects compliant to **A** ⇔ **O** is an **A_IMPL** class object*

We continue our discussion on class objects by presenting the implementation (in the Delta language) of a generic function, which accepts a set of base class objects and a derived mixin class object, and performs mixin composition returning the new composite class object.

Dynamic mixin composition of class objects

Mixin inheritance [Bracha90] concerns base class parameterization resulting in generic derived classes. It is also called inheritance on demand or genericity [Meyer97], since for mixin class *B*, the base class *T* is a parameter to the compile-time composition operator *B[T]*, without strong coupling among *B* and the classes "similar" to *T*. Following our previous discussion, the mixin composition operator on class objects should return a class object producing instances compliant to the mixin composition of the respective classes.

```
    function mixin_comp() {
      local n = tablength(arguments);
      comp = [
        {.B     : arguments[0] },  ←The B class object
        {.n     : n - 1 },  ←Stores number of base A_i class objects
        {.new   : (method(){  ←Object constructor for the composite class object
          args  = [];
          argNo = 0;
          if (arguments[0] == self.B.class)
               { args = arguments[1]; argNo = 2; }
          B_obj      = self.B.new(|args|);  ←Construct a B object
          B_obj.n    = self.n;  ←Store the number of total base objects
          for (i=1; i <= self.n; ++i) {  ←Instatiate all base objects
               args  = [];
               Ai    = self["A" + i];  ←Get the A_i class object
               if (arguments[argNo] == Ai.class)
                    { args = arguments[++argNo]; ++argNo; }
               Ai_obj = Ai.new(|args|);  ←Construct an A_i object
               inherit(B_obj, Ai_obj);  ←Establish inheritance B ↙ A_i
               B_obj[Ai.class] = Ai_obj;  ←Store the produced base object
          }
          return B_obj;  ←Return the B (most-derived) object
        })}
      ];
      for (i = 1; i <= n - 1; ++i)
           comp["A"+i]=arguments[i];  ←Store base class objects as keys "Ai"
      return comp;  ←This is the class object pertaining to the composition of mixins.
  }
```

Figure 11: Mixin composition among class objects in Delta; the key point is the implementation of the new function of the composite class object producing objects compliant to $B[A_1...A_k]$.

For example, let's assume the class objects *A_IMPL* for class *A*, and *B_IMPL* for mixin class *B*. Then, the mixin composition among *A_IMPL* and *B_IMPL* is any class object producing objects compliant to *B[A]*. We consider the general form of an *N*-ary mixin composition operator among a mixin class *B* and a sequence of base classes $A_1,\ldots,A_k$ $K\geq 1$ evaluating as $B[A_1,\ldots,A_k]$. We implement this operator in the Delta language, as a function over *B* and A_i class objects, returning a new class object producing objects compliant to $B[A_1,\ldots,A_k]$. The implementation is provided in Figure 11. The expression *arguments* is a read-only table carrying all actual arguments as $arguments[i]^{\,i:0\ldots N-1}$ while *tablength* is a library function returning table size *N*. Also, for a table *P* with *N+1* elements indexed as *P[0],…,P[N]*, the expression |*P*| as an actual argument evaluates by pushing onto the stack *P[0],…,P[N]* as if they those were explicitly supplied arguments. Following Figure 11, the function *mixin_comp* returns an object *comp* with a method named *new* producing objects reflecting the mixin class inheritance scheme – see also Figure 12. Inside *new*, two basic steps are taken: (i) an object *B_obj* complying to *B* class is constructed; and (ii) for each A_i a base object *Ai_obj* is constructed, set as the base of *B_obj* via an *inherit* call.

Actual argument propagation There is one remaining issue to be explained. For any *C* returned class object with *C.new* constructor for creating composite objects, graceful initialization of the *B* and A_i constituent objects should be facilitated. In general, there is no information regarding the *B* and A_i constructor signatures, meaning we should accommodate all construction possibilities for the involved class objects. An impractical solution is to rely on default constructors, requiring programmers to explicitly bring composite objects, after construction with no parameters, to the desirable state. In our implementation, the *new* method of the returned class object adopts a parameter passing pattern for propagating actual arguments to the corresponding constituent-object constructor.

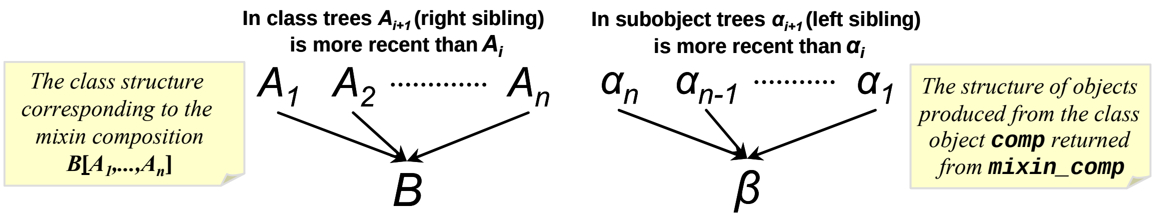

In class trees A_{i+1} (right sibling) is more recent than A_i
In subobject trees α_{i+1} (left sibling) is more recent than α_i
The class structure corresponding to the mixin composition $B[A_1,\ldots,A_n]$
A_1 A_2 A_n
B
α_n α_{n-1} α_1
β
The structure of objects produced from the class object comp returned from mixin_comp

Figure 12: Mixin class structure (left) and object structure produced by the comp class object (right).

We give two examples before generalizing on the call pattern. Consider the composition among classes *B* and $A_1,\ldots,A_k$ as before, returning the class object *C*. Lets assume that the call *C.new(x,y,z)* should construct a composite object with *(x, y, z)* passed for construction of the *B* object, having every A_i created with its default constructor. Then, instead of *C.new(x,y,z)*, the call should be made as *C.new("B",* **[***x,y,z***]***)* - see *Example 1* of Figure 13. Similarly, *C.new(10, x,y,z, f(a))*, with *10* for *B*, *(x,y,z)* for A_1 and *f(a)* for A_7, should be made as *C.new("B",* **[***10***]**, "A_1", **[***x,y,z***]** "*A7*", **[***f(a)***]***)* - see *Example 2* of Figure 13.In general, the call to *new* is made following the pattern at the bottom left of Figure 13. The prerequisite is that involved class objects encompass a member "*class*" with a unique string value among them. The *[]* means optional while **[]** is array construction. Following this pattern, assuming A_0 equivalent to B, the list of actual arguments is made of optional ordered pairs $<Name_i, Array_i>$, $i{:}0...K$, where if $Name_i{=}A_i.class$ then $Array_i$ holds the arguments for constructing A_i via $A_i.new(|Array_i|)$. The latter relates to the calls *self.B.new(|args|)* and *Ai.new(|args|)* of Figure 13.

Example 1:

C.new("*B*", [*x*,*y*,*z*]**)** $\Rightarrow$

B**.new(***x, y,z***)**

A_1**.new()**

…

A_N**.new()**

With A_i ***equivalent to*** B ***the generic call pattern is:***

C.new($[$ "A_i", **[** A_i *args* **]** $]^{i:0...K}$ **)**

Example 2:

C.new("*B*", **[***10***]**, "A_1", [*x*,*y*,*z*] "*A7*", [*f(a)*]**)** $\Rightarrow$

B**.new(***10***)**

A_1**.new(***x, y, z***)**

A_2**.new()**

…

A_7**.new(***f(a)***)**

…

A_N**.new()**

Figure 13: The call pattern for the new method of the class object resulting from mixin composition.

8 SUMMARY AND CONCLUSIONS

We presented a new form of dynamic untyped object-based inheritance, as implemented in the context of a language named Delta, relying on object trees and a member resolution algorithm guaranteeing late-binding of inquired members to their most-recent version. The proposed programming model emphasizes inheritance associations adopting the metaphoric notions of 'base' and 'derived', offering a small set of inheritance control functions for dynamic inheritance and un-inheritance. Through a case scenario we show how the proposed inheritance model is: (a) practically superior to embedding, as the latter is an excessively complicated programming model for uninheritance and non-monotonic evolution of objects; and (b) more appropriate than delegation, since, for certain generic scenarios, delegation fails to support polymorphism due its parental lookup policy. Our approach of explicitly introducing metaphoric notions of inheritance in an untyped inheritance context, via a lookup algorithm preserving monotonicity at the metaphoric semantic domain, reflects a programming model resembling that of class-based inheritance, however, without compromising the compositional flexibility of untyped object-based inheritance.

We consider the latter design decision as particularly advantageous. By adopting standard notions of typed inheritance we offer a programming model relatively familiar to programmers of class-based languages. Although such metaphoric notions are not mapped to a type system, they are 'semantically explicit', since they are reflected in the inheritance control functions and the member lookup algorithm. This feature makes transitions among class-based and classless languages more natural, even straightforward, in comparison to delegation and embedding languages.

We argue that untyped object-based inheritance deserves more attention, primarily due to the increased possibilities for compositional schemes through inheritance. Classes and inheritance always remain the primary design concepts, usually implemented in most languages around prototype frameworks. As early identified [Smith94], overemphasis on technical issues has led to numerous solutions reflecting diverse programming models. Till today, little progress has been made towards modeling and composition patterns over prototypes, while the impression that prototypes are prototypical objects is still around. We have discussed why the latter is a serious design misconception, demonstrating how dynamic mixin composition on class objects is easily implemented through the proposed inheritance framework.

REFERENCES

[Abadi96] Martin Abadi, Luca Cardelli: 'Object Protocols', Chapter 3.5, 'A Theory of Objects', Springer-Verlag, pp. 32-33, 1996.

[Agesen00] Ole Agesen, Lars Bak, Craig Chambers, Bay-Wei Chang, Urs Hölzle, John Maloney, Randall Smith, David Ungar, Mario Wolczko: 'The SELF 4.1 Programmers Reference Manual', 2000.

[Anderson02] Chris Anderson, Sophia Drossopoulou: 'δ: an imperative object based calculus with delegation', Technical report available on-line from: http://www.cee.hw.ac.uk/DART/publications/auto/And+Dro:delta-2002.html, 2002.

[Blaschek91] Gunter Blaschek: 'Type-Safe Object-Oriented Programming with Prototypes - The Concepts of Omega', *Structured Programming* Journal, Vol. 12, No 1, pp. 217-226, 1991.

[Bracha90] Gillad Bracha, William Cook: 'Mixin-based inheritance'. In proceedings of the ACM ECOOP/OOPSLA'90 conference, 1990, pp 303 - 311.

[Cardelli95] Luca Cardelli: 'A language with distributed scope', *ACM POPL'95* proceedings, pp. 286 – 297, 1995.

[Cardelli96] Luca Cardelli: 'Object-based vs class-based languages', *ACM PLDI'96* Tutorial, 1996.

[Ierusalimschy03] Romeo Ierusalimschy: 'Programming in Lua', Book available on line from http://www.lua.org/pil/, ISBN 85-903798-1-7, 2003.

[Kniesel00] Gunter Kniesel: 'Darwin -- Dynamic Object-Based Inheritance with Subtyping'. *PhD thesis*, CS Dept. III, University of Bonn, Germany, July 2000.

[Lieberman86] Henry Lieberman: 'Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems', *ACM OOPSLA'86* Proceedings, pp. 214-223, 1986.

[Liskov87] Barbara Liskov: 'Data Abstraction and Hierarchy', *ACM OOPSLA'87*, keynote, in ACM SIGPLAN Notices, Vol. 23, No. 5 (May 1987), pp. 17-34, 1987.

[Meyer97] Bertrand Meyer: 'Object-Oriented Software Construction', Second Edition, Prentice Hall, Santa Barbara, CA, 1997.

[Ousterhout98] John K. Ousterhout: 'Scripting:Higer-Level Programming for the 21st Century', *IEEE Computer*, March 1998, pp. 23-30, 1998.

[Smith94] Randall Smith, Lentczner Mark, Walter R. Smith, Antero Taivalsaari, David Ungar: 'Prototype-based languages: object lessons from class-free programming', Panel in *ACM OOPSLA'94* Proceedings, pp. 102-112, 1994.

[Smith95] Randall B. Smith, David Ungar: 'Programming as an Experience: The Inspiration for Self', *ECOOP'95* Proceedings, Springer LNCS Vol. 952, pp. 303-330, 1995.

[Stein87] Lynn Andrea Stein: 'Delegation is Inheritance', *ACM OOPSLA'87* Proceedings, pp. 138-146, 1987.

[Taivalsaari93] Antero Taivalsaari: 'A critical view of inheritance and reusability in object-oriented programming', PhD Thesis, Jyvaskyla Studies in Computer Science, Economics and Statistics 23, University of Jyvaskyla. Finland, ISBN 951-34-0161-8, 1993.

Appendix

The *isderived* function

```
isderived (a, b) {                    Returns if a←b
  if b.Derived = a then               Check if a↙b holds (i.e. directly derived)
      return true
  else {
      for each x in a.Bases do        Check if any of the base subobjects is derived from b
          if isderived(x, b) then
              return true
      return false
  }
}
```

The Delta language syntax

```
code          ::= { [ def ] }
def           ::= ( [ stmt ] ';' | func )
func          ::= ( funcprefix | methodprefix ) funcdef
funcprefix    ::= 'function' [ id ]
methodprefix  ::= 'method'
funcdef       ::= '(' [ id { ',' id } ] ')' block
block         ::= '{' code '}'
funcexpr      ::= '(' func ')'
stmt          ::= (  expr | whilest | forst | ifst | 'break' | 'continue' |
                     'return' [ expr ] | block | assertion |
                     'const' id '=' expr | 'try' stmt 'trap' lval stmt |
                     'throw' expr )
whilest       ::= while '(' expr ')' stmt | 'do' stmt 'while' '(' expr ')'
forst         ::= 'for' '(' exprlist ';' expr ';' exprlist ')' stmt
ifst          :: 'if' '(' expr ')' stmt [ 'else' stmt ]
exprlist      ::= [ expr { ',' expr } ]
expr          ::= ( assign | primary | boolean | arith )
assign        ::= (  lval '=' expr | lval '+=' expr | lval '-=' expr |
                     lval '*=' expr | lval '/=' expr )
lval          ::= ( [ 'local' | 'static' | 'global' | '::' ] id | member )
member        ::= ( expr get id | expr subscr | expr get string )
```

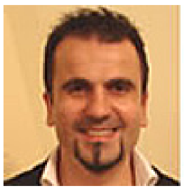
```
get           ::= ( ‘.’ | ‘..’)
subscr        ::= ( ‘[’ expr ‘]’ | ‘[[’ expr ‘]]’ )
primary       ::= (  lval | lval (‘++’|‘--') | (‘++’|‘--') lval | ‘lambda’ |
                    const | callable )
                    object | ‘self’ | ‘arguments’ | ‘-’ expr | ‘not’ expr )
callable      ::= ( lval | ‘(’ expr ‘)’ | funcexpr | call )
call          ::= callable ‘(’ [ actual  { ‘,’ actual } ] ‘)’
actual        ::= ( expr | ‘|’ expr ‘|’ )
const         ::= ‘nil’ | ‘true’ | ‘false’ | number | string
boolean       ::= expr boolop expr
arith         ::= expr arithop expr
arithop       ::= ‘+’ | ‘-‘ | ‘*’ | ‘/’ | ‘%’
boolop        ::= ‘or’ | ‘and’ | ‘<’ | ‘>’ | ‘<=’ | ‘>=’ | ‘==’ | ‘!=’
object        ::= ‘[’ [ slot { ‘,’ slot} ] ‘]’
slot          ::= ( ‘{’ expr {‘,’ expr } ‘:’ slotval ‘}’ | slotval )
slotval       ::= ( expr | methoddef )
methoddef     ::= ‘(’ ‘method’ funcdef ‘)’
```

assertion ::= ‘assert’ expr

Availability information

The Delta language (compiler, standard libraries, virtual machine), with the full examples presented in this paper, are available (Windows only) via anonymous ftp from the following address: http://www.ics.forth.gr/hci/files/plang/DELTA.ZIP. The Delta IDE named Sparrow is available from http://139.91.186.232/sparrow-setup.exe (installer, Windows only). The latter are only an initial packaging to allow programmers or language developers have a hands-on experience with the language and it’s IDE. Once the site for Delta is built, the Delta ftp address will contain the necessary redirection information.

About the author



Anthony Savidis is an Associate Professor of ‘Programming Languages and Software Engineering’ at the Department of Computer Science, University of Crete, and the Technical Coordinator of the HCI Laboratory, Institute of Computer Science - FORTH. His e-mail address is as@ics.forth.gr