Efficient sideways composition in COLAs via 'Lieberman' prototypes

Ian Piumarta

VPRI Memo M-2007-002-a

Viewpoints Research Institute, 1209 Grand Central Avenue, Glendale, CA 91201 t: (818) 332-3001 f: (818) 244-9761

Efficient sideways composition in COLAs via 'Lieberman' prototypes

$Id: prototypes.html.in 2 2007-02-22 16:58:31Z piumarta $
last updated for the idst-5.8 release

Contents:

1 Introduction

This memo describes the addition of sideways composition to COLA objects in the form of 'Lieberman-style' prototypes.

1.1 Inheritance and delegation

Every COLA object is created as a member of a particular *clone family*. A clone family provides all its members with their behaviour and dictates their internal layout. Modifying the behaviour of one clone within a family causes the new behaviour to be adopted by all members of the family. (Modifying the layout of a clone is effectively impossible once the initial 'examplar' of the family has been defined.) Since the layout of a clone is intimately related to the methods that are allowed to run 'in' that clone (such as slot getters and setters), the patterns of communication between clone families within an inheritance hierarchy are rigid and fixed at the point of their definition. This is why inheritance occurs between *vtables* (each a method dictionary shared by all members of a family) in COLAs.

In Lieberman prototypes, delegation occurs between objects (not between object behaviours). Any object can delegate messages to any other object at any time. The entire chain of delegation between any number of prototypes is considered a single *composite object*. The meaning of `self` is independent of the number of times a message has been delegated between prototypes since the original send. Sending to `self` from any prototype within the composite object causes the dispatch to begin again in the 'outermost' prototype.

Compared to inheritance, delegation is the more flexible and general of the two techniques. However, they both have their place within an object model: inheritance for sharing of implementation state (and the methods that act upon it) for a single prototype (within a hierarchy of related prototype families), and delegation for sideways composition of (independent and previously unrelated) prototypes into a single logical composite object. This is the position adopted (and implemented) for sideways composition of COLA objects.

2 Implementation

Two central concepts: receiver splitting and multi-dimensional dynamic binding.

2.1 Receiver splitting

The `self` used for sending messages is dissociated from the `self` used for accessing state. The first identifies the entire composite object and is always the first prototype in the delegation chain regardless of how many times a given message has been delegated within that chain. The second identifies the particular prototype (within the delegation chain) associated with the currently executing method (physically holding the instance variables visible to that method).

COLA method signatures are augmented accordingly. A method

```
FamilyName messageName: arguments... [ ... ]
```

previously declared by the compiler as

```
oop FamilyName__messageName(oop closure, oop self, arguments...) { ... }
```

is now declared as

```
oop FamilyName__messageName(oop closure, oop stateful_self, oop self, arguments...) { ... }
```

Within this method, sends of the form

self selector: arguments...

are compiled (as usual) as

```
_send(s_selector, self, arguments...)
```

whereas accesses to named instance variables that were previously compiled as

```
self->v_instVarName
```

are now compiled as

```
stateful_self->v_instVarName
```

The Lieberman-style prototypical delegation described above can be effected by causing `self` and `stateful_self` to diverge. The former remains constant while the latter moves through the delegation chain to record the particular prototype in which delegated method lookup succeeded for the currently executing method.

Both `self` and `stateful_self` are treated by the compiler as normal method arguments. If 'self' appears free in a lexically-enclosed block then `self` will be assigned a slot in the state vector of the block's defining context that is stored in the closures created each time control passes the block's point of definition in the program. Similarly, if an instance variable appears free in an inner block then `stateful_self` will be assigned to a slot in the state vector, and hence captured and stored in all closures associated with the block. This guarantees correct (and intuitively expected) behaviour of sends to 'self' and accesses to state within arbitrarily-nested blocks of prototypical methods.

2.2 Multi-dimensional dynamic bind

The usual *inheritance chain* (between clone families) is augmented by an orthogonal *delegation chain* (between arbitrary objects).

Message sends were previously implemented by code equivalent to

```
#define _send(MSG, RCV, ARG...) ({              \
  register oop _r= (RCV);                       \
  struct __closure *_c= _bind((MSG), _r);       \
  (_c->method)((oop)_c, _r, ##ARG);             \
})
```

in which `_bind()` returns a single result: the closure in which the method's implementation address is stored. In order to support prototypes the `bind()` function is augmented to return two results: the closure in which the method implementation is stored and the prototype in which the method binding was found.

```
struct __lookup {
  struct __closure *closure;
  oop               prototype;
};

#define _send(MSG, RCV, ARG...) ({                                      \
  register oop _r= (RCV);                                               \
  struct __lookup _l= _bind((MSG), _r);                                 \
  (_l.closure->method)((oop)_l.closure, _l.prototype, _r, ##ARG);       \
})
```

The last line passes the closure (as before) followed by the prototype in which the bind succeeded (which becomes the `stateful_self` in the invoked method) and the original receiver (which becomes `self` in the invoked method and is the receiver for all sends to 'self').

The implementation of `_bind()` is extended from the original one-dimensional lookup

```
struct __closure *_bind(oop selector, oop receiver)
{
  if (cache[receiver.vtable].selector == selector)
    return cache[receiver.vtable].closure;
  assoc := receiver.vtable.lookup(selector);
  if (assoc == nil)
    errorDoesNotUnderstand();
  cache[receiver.vtable].selector= selector;
  return cache[receiver.vtable].closure= assoc.closure;
}
```

to search the inheritance chain (as before) for each prototype in the delegation chain:

```
struct __lookup _bind(oop selector, oop receiver)
{
  do {
    if (cache[receiver.vtable].selector == selector)
      return (struct __lookup){ cache[receiver.vtable].closure, receiver };
    assoc := receiver.vtable.lookup(selector);
    if (assoc != nil) {
      cache[receiver.vtable].selector= selector;
      cache[receiver.vtable].closure= assoc.closure;
      return (struct __lookup){ assoc.closure, receiver };
    }
    receiver := receiver._delegate();  /* message send */
  } while (receiver != nil);
  errorDoesNotUnderstand();
}
```

Note that the delegate for a given prototype is obtained by sending it a '_delegate' message. The delegation chain can be defined by state (have the method answer an instance variable) or by computation (have the method compute and answer the desired next prototype object in the delegation chain). The default implementation of '_delegate' installed in '_object' simply answers 'nil'.

The overhead (once caches have settled) for this implementation strategy is (on average) one method cache probe and one message send for each step along the delegation chain. The overhead for objects that do not participate in prototypical delegation is essentially zero: their '_delegate' method always returns `nil` and is only ever invoked immediately before a guaranteed 'doesNotUnderstand' situation. No state whatsoever is added to these objects.

The implementation overhead was a couple of tens of lines of code.

3 A short example

```
Prototype : Object ( next )

Prototype _delegate [ ^next ]

Prototype withDelegate: anObject
[
    self := self new.
    next := anObject.
]

A : Prototype ()

A a     [ 'A.a' putln ]

B : Prototype ()

B a     [ 'B.a' putln ]
B b     [ 'B.b ' put.  self a ]

C : Prototype ()
```

```
C c     [ 'C.c ' put.  self a; b ]

[
    | a |
    a := A withDelegate: (B withDelegate: C new).
    '======== a a:\n' put.  a a.
    '======== a b:\n' put.  a b.
    '======== a c:\n' put.  a c.
]
```

Executing the above program generates the following output:

```
======== a a:
A.a
======== a b:
B.b A.a
======== a c:
C.c A.a
B.b A.a
```

4 Caveats

Assigning to 'self' causes both `self` and `stateful_self` to be assigned (the latter is 'tied' to the former for the purposes of assignment). This is necessary to guarantee correct behaviour in 'constructor' methods.

The default definition of 'new' clones only the first prototype in the delegation chain. This method must be overridden if all (or some arbitrary portion) of the delegation chain is to be duplicated when cloning the 'composite object' formed by them.

I haven't (yet) figured out a syntax for 'resend', analagous to 'super' but for the delegation chain (rather than the inheritance chain).

5 Applications

Sideways composition. (As described above.)

Bullet-proof proxies, formed by a two-deep delegation chain in which the first prototype is the proxy and the second is the object 'wrapped' by the proxy. All sends to `self` within methods of the wrapped object will be invoked on the enclosing proxy object.

Layers and Context-Oriented Programming. Supporting these effectively was the principal motivation for adding 'Lieberman' prototypes to COLAs.

Implementation object for JavaScript, Python and similar languages.

Using 'here sends' to implement robust proxies (methods that cannot be overriden in a more specific prototype within the composite object).

'Perfect' encapsulation: no state whatsoever is visible between prototypes. Message sends are the only form of interaction between prototypes.

Mixins and supporting mechanism for traits, multiple inheritance, etc.

6 Acknowledgements

Robert Hirschfeld and Michael Haupt of the Hasso-Plattner-Institut contributed significantly to several days of fascinating coversation and debate about mechanisms suitable for supporting 'layers' and 'context-oriented programming' that led to the implementation described in this memo. Henry Lieberman's revolutionary writings on composite objects provided the simplest and most powerful model that could be adopted as a basis for that support.