Open, extensible object models

Ian Piumarta
Viewpoints Research Institute
9242 Beverly Blvd, Suite 300
Beverly Hills, CA 90210, USA
ian@vpri.org

Alessandro Warth
UCLA Computer Science Department
3440 Boelter Hall
Los Angeles, CA 90095, USA
awarth@cs.ucla.edu

Abstract

Programming languages often hide their implementation at a level of abstraction that is inaccessible to programmers. Decisions and tradeoffs made by the language designer at this level (single vs. multiple inheritance, mixins vs. Traits, dynamic dispatch vs. static case analysis, etc.) cannot be repaired easily by the programmer when they prove inconvenient or inadequate. The artificial distinction between implementation language and end-user language can be eliminated by implementing the language using only end-user objects and messages, making the implementation accessible for arbitrary modification by programmers. We show that three object types and five methods are sufficient to bootstrap an extensible object model and messaging semantics that are described entirely in terms of those same objects and messages. Raising the implementation to the programmers' level lets them design and control their own implementation mechanisms in which to express concise solutions and frees the original language designer from ever having to say "I'm sorry".

1. Introduction

Most programming languages and systems make a clear distinction between the *implementation level* in which the system is built and the *'end-user' level* in which programs are subsequently written. The abstractions and semantics provided by these programming systems are effectively immutable. Metaobject Protocols (MOPs) [5] are designed to give back some power to programmers, letting them extend the system with new abstractions and semantics. We are interested in a different approach to solving the same problem where we eliminate the distinction between the implementation and user levels of the programming system.

As an example of the problem we are trying to solve, consider the implementation of a Lisp-like language with several atomic object types. The implementer must choose a representation for these objects in some (typically lower-level) implementation language. The choice of representation can have a profoundly limiting effect on the ability of both the implementer and end-user to extend the language with new types, primitive functionality and semantics at some later time. Our Lisp-like end-user language might have C as its implementation language and use a *discriminated union* to store atomic objects and 'cons' cells:

```
enum ObjectTag { Number, String, Symbol, Cons };

struct Object {
  enum ObjectTag  tag;
  union {
    struct Number number;
    struct String string;
    struct Symbol symbol;
    struct Cons   cons;
  } payload;
};
```

With this representation, each primitive in the end user language that manipulates data would use conditional (`if` or `switch`) statements to select appropriate behaviour according to the `tag` field.

This simple object model has already made significant design decisions and rendered them immutable:

- All objects must start with an integer `tag` field.
- The internal layout of the four intrinsic types cannot be modified at runtime.

The consequences of these decisions include:

- New payloads cannot be added by end user code, especially if they require more storage than the intrinsic types.
- New tags cannot be added unless all primitives are explicitly designed to work in the presence of arbitrary tags, or the user is in a position to understand, modify and then recompile every part of the base language implementation that might be concerned with object tags.

We could start to address these problems by creating a more general object model for our structured data, for example by

adding a `size` field to allow for arbitrary payloads. Unfortunately each such change *adds* complexity to the language runtime and imposes *more* 'meta-structure' in the objects, ultimately making them *less* amenable to unanticipated deep modifications in the future.

These problems are even more severe when we consider object-oriented languages. The object model for a simple prototype-based language might specify 'method dictionary' and 'parent' slots in every object. The runtime would look up a message name in the receiver's `methodDictionary`, trying again in the `parent` object's method dictionary if no match is found, continuing until finding a match or reaching the end of the parent chain. Adding multiple delegation to this language would be difficult because the runtime assumes that the `parent` field contains a single object and not, for example, a list of parent objects to try in turn.

The trouble is that some of the semantics of the above example (single delegation between instances) are reified eagerly in the execution mechanisms of the language. This in turn eagerly imposes supporting meta-structure (instances chained through a `parent` slot) within the objects. Since the execution mechanisms are expressed in an implementation language at a lower level of abstraction than that of the end user language, neither the mechanisms nor their effects on object structure can be modified by end users. Moreover, adapting the implementation machinery for reuse in supporting a different end-user language is more difficult when the required changes are pervasive and expressed in a low-level implementation language. In this paper we present an object model intended to eliminates most of these problems:

- We show how an object-based model of data can help alleviate some of the problems of extensibility in programming language implementation (Section 2).
- We define a simple, extensible object model that imposes no structure on end-user objects (Section 3).
- The end-user object model provides message-passing semantics implemented using its own objects and messaging mechanism, making the semantics of messaging modifiable or even replaceable from within the end-user language. We show that three kinds of object and five small methods are sufficient to achieve this (Section 3.1).
- The flexibility gained by exposing the object model's semantics is illustrated by showing that it can be extended easily to support language features including multiple inheritance and mixed-mode execution [10] (Sections 2.2 and 3).
- We validate the use of this approach for production systems by showing that: it has low space overhead (Section 5); its performance can be competitive with, and in some cases even better than, equivalent 'static' implementation techniques (Section 5.3); existing object models can be easily implemented on top of our model (Section 5.1); advanced compositional techniques such as Traits [11] can be accommodated (Section 5.2).

2. Our object model by example

Our model describes one thing: how an object responds to a message. Each object is associated with a *vtable* object. When a message is sent to an object O, its vtable V is asked to find the appropriate method to run. This is done by sending the message '`lookup`' to V, with the message name as argument. The semantics of sending a message to O are therefore determined entirely by V's response to the '`lookup`' message. By overriding (or redefining) '`lookup`' we can change the semantics of message sending for some (or all) objects.

The vtable object doesn't have to be a table. It can determine the method to run for a given message send in any way it wants. Often, though, vtables are simply dictionaries mapping message names onto method implementations.

This section introduces our our object model by using it to solve two of the problems mentioned in the introduction: adding a new atomic object type to a Lisp-like language and converting single delegation to multiple delegation in a message-passing language.

2.1 Adding data types to a language

For our Lisp-like language we might have a `length` primitive that tells us how many elements are present in a string or list. Using the `tag` field in the `Object` structure to discriminate the type of payload, `length` might look like this:

```
int length(struct Object *object)
{
  switch (object->tag)
  {
    case Number: error("numbers have no length");
    case String: return object->payload.string.length;
    case Symbol: error("symbols have no length");
    case Cons:   return object->payload.cons.cdr
                ? 1 + length(object->payload.cons.cdr)
                : 1;
    default:     error("illegal tag");
  }
}
```

Let's add a vector type to this language. We have to extend the above `switch` statement with a new `case` to take into account our new data type and its `tag` value:

```
    case Vector: return object->payload.vector.length;
```

This isn't too bad if we are the only user of the language and we have access to the source code of the implementation. However, the situation is much worse if we want to share the new type with other users of the language, possibly as a third-party extension; any primitive that is not modified with an additional `case` to handle vectors will cause a run-time error.

It would be better to store the relevant `case` implementation from each primitive function in the data type itself. Using our object model the new data type is added to the

```
struct vtable *Vector_vt = 0;

int Vector_length(struct Vector *vector) {
  return vector->length;
}

void initialise(void) {
  ...
  Vector_vt = send(vtable, s_allocate,
                   sizeof(struct vtable));
  send(Vector_vt, s_addMethod, s_length, Vector_length);
  ...
}

int length(struct object *object) {
  return send(object, s_length);
}
```

Figure 1. Creating a new type and associating functionality with it. The vtable `Vector_vt` describes the behaviour of our new type. Invoking the method `s_addMethod` in it makes an association between the selector `s_length` and the method implementation `Vector_length`. Our `length` primitive can now simply invoke the method `s_length` in any object and expect it to respond appropriately regardless of the number of data types supported by—or added to—the language. (The variables prefixed with `s_` are symbols: interned, unique strings suitable for identifying method names.)

language by creating a new vtable (object behaviour) and then installing its primitives as methods in the vtable. Figure 1 shows what this would look like in our object model, again using C as the implementation language.

This is more than advocating an object-oriented style of programming language construction. Consider the same Lisp-like language implemented in C++. Even if the `length` primitive was made a virtual function of each supported data type, we would have to recompile every file after adding `Vector` since the layout of C++ vtables is computed at compile time; adding a new virtual method would invalidate all previous assumptions about the vtable layout.

Perhaps more compelling is an example involving an object-oriented language that uses our object model and that can modify the direct semantics of its own messaging mechanism.

2.2 Adding multiple inheritance to a prototype-based language

We have created a high-level, prototype-based programming language with single delegation that uses our object model directly for its end user objects.[1] We will use this language for several examples. Its syntax is very close to that of Smalltalk [4] with a few small differences (described in Appendix A).

[1] This language is written entirely in itself and can be downloaded, along with many examples including those presented in this paper, from `http://piumarta.com/oopsla07`

Everything in our object model is an object, including the vtables that describe the behaviour of objects. Interacting with vtables is just a matter of invoking methods in them. One such method is called `lookup`; it takes a method name as an argument and returns a corresponding method implementation. By overriding (or redefining) this method we can change the semantics of message sending for some (or all) objects.

Our prototype-based language provides the programmer with single inheritance; a given *family* of objects *inherits* behaviour from a parent family (with all families eventually inheriting behaviour from `Object`). Figure 2 shows how the programmer can directly add multiple inheritance to this language, without loss of performance.[2] With these additions to the language, and given three prototype families `C1`, `C2` and `C3`

```
C1 : Object ()
C1 m  [ 'this is m' putln ]

C2 : Object()
C2 n  [ 'this is n' putln ]

C3 : C1 ()  "C3 inherits from C1"
```

the programmer can now dynamically add `C2` as a parent of `C3`

```
C3 vtable addParent: C2 vtable
```

so that objects in its family can execute methods inherited from both `C1` and `C2`:

```
C3 new
  m;  "inherited from C1"
  n   "inherited from C2"
```

A serious implementation would of course have to take state and behavioural conflicts into account, although this could be as simple as allowing only one parent to be stateful and disallowing duplicated message names. (Our implementation of Traits [11] in Section 5.2 illustrates this.)

3. Open, extensible object models

An object typically describes both *state* and *behaviour* that acts on (or is influenced by) that state. We might account for both state and behaviour in our object model, but it would be simpler to model just one of them and then use it to provide the other indirectly. We choose to model (and expose) behaviour as a set of *methods* that are invoked in an object by name; access to state, if appropriate, is then provided through 'accessor' methods.[3]

Figure 3 illustrates this simple model: an object is some opaque quantity in which a method can be invoked by name;

[2] The message sending mechanism uses a *method cache* to memoize the result of invoking `lookup` in a given `vtable` for a given `messageName`. The overhead of iterating through multiple parents is incurred only when the method cache misses, which is rarely [2].

[3] The discussion of related work (Section 6) mentions Self, a system that made the opposite choice of modeling behaviour as a special kind of state.

```
ParentList : List ()

vtable addParent: aVtable
[
  parent isNil
    ifTrue: [parent := aVtable]
    ifFalse:
     [parent isParentList
        ifTrue:  [parent add: aVtable]
        ifFalse: [parent := ParentList new
                    add: parent;
                    add: aVtable;
                    yourself]]
]

ParentList lookup: messageName
[
  | method |
  self do: [:aVtable |
    (method := aVtable lookup: messageName) notNil
      ifTrue: [↑method]].
  ↑nil
]
```

Figure 2. Adding multiple inheritance to a prototype-based language. We will store multiple parents in `ParentList` objects; these extend (inherit behaviour from) `List` without adding any additional state. We tell `vtable` how to `addParent:` by converting a single parent `vtable` into a `ParentList` if necessary, then adding the new parent `vtable` to the list. Next we define `lookup:` for `ParentList` to search for the `messageName` in each parent consecutively. (The `lookup:` method already installed in `vtable` can be left in place; it performs a depth-first search up the inheritance chain by invoking `lookup:` in its `parent` slot, which can now be either a `vtable` or a `ParentList`.)

we call the set of methods associated with a given object its *behaviour*. Since we wish to avoid imposing structure on end user objects, the description of behaviour is stored separately from the object in a manner similar to most object-oriented languages; in particular, `parent` slots and method tables are not stored in objects. An object is therefore a *tuple* of behaviour and state. Since the behaviour is decoupled from the internal state of the object it can be replaced and/or shared as desired, or even associated implicitly with the object.[4]

Figure 4 shows the layout of our objects in memory. An ordinary object pointer (oop) points to the first byte of the object's internal state (if any). The object's behaviour is described by a *virtual table* (vtable). A pointer to the vtable is placed immediately before the object's state, at offset -1 relative to the pointer. This is done to preserve pointer identity for objects that encapsulate a foreign structure, facilitating communication with the operating system and libraries. It

[4] In our prototype language, tagged (odd) pointers and the null pointer are implicitly associated with vtables for the behaviour of small integers and nil, respectively.

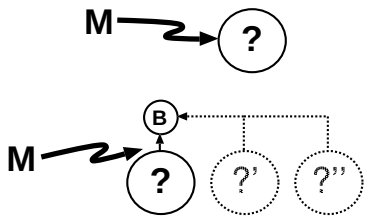

Figure 3. Minimal object model. An object is some opaque state *?* on which a method M can be invoked by name. To implement this model we need a mapping from method names to method implementations. So, to invoke a method M in the object *?* we find the corresponding method implementation in a behaviour description B. An object is therefore a tuple of behaviour B and state *?*. Since behaviour is separate from the object it describes, it is possible to share any given behaviour B between several distinct objects *?*, *?'*, *?''*, ...

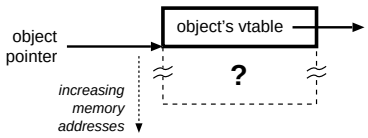

Figure 4. Implementation of minimal object. An object pointer (oop) points to the start of the object's internal state (if any). The object's behaviour is described by a *virtual table* (vtable). A pointer to the vtable is placed one word before the object's state.

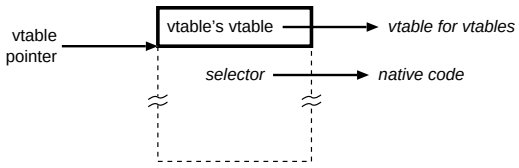

Figure 5. Internals of vtables. A vtable maps a message name (selector) onto the address of the native code that implements the corresponding method. The mapping is determined by the vtable's response to the `lookup` message, which is bound to an implementation by the 'vtable for vtables'.

also allows compiled methods, identified by the address of their first instruction, to be full-fledged objects.

A vtable is an object too, as shown in Figure 5, and has a reference to the 'vtable for vtables' before its internal state. This 'vtable for vtables' is its own vtable, as shown in Figure 6. It provides a default implementation of the `lookup` method (for all vtables) that maps message names onto method implementations. The state within a vtable supports this mapping. The `lookup` method therefore dictates the internal structure of all vtables, but there is nothing spe-

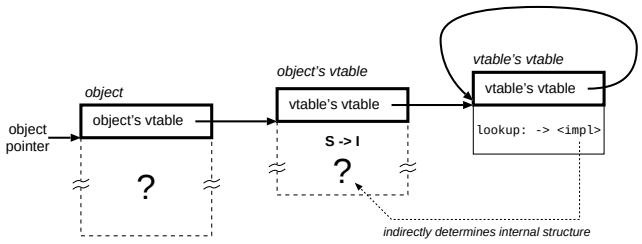

Figure 6. Everything is an object. Every object has a vtable that describes its behaviour. A method is looked up in a vtable by invoking its `lookup` method. Hence there is a 'vtable vtable' that provides an implementation of `lookup` for all vtables in the system, including for itself. The implementation of this `lookup` method is the only thing in the object model that imposes internal structure on vtables.

type	*method*
object	
symbol	intern
vtable	addMethod
vtable	lookup
vtable	allocate
vtable	delegated

Table 1. Essential objects and methods. For `vtables`, `addMethod` creates an association from a message name to a method implementation, `lookup` queries the associations to find an implementation corresponding to a message name, `delegated` creates a new `vtable` that will delegate unhandled messages to the receiver, and `allocate` creates a new object within the `vtable`'s family (by copying the receiver into the new object's `vtable` slot). We include `symbol`'s `intern` method in this list since the end user must have some way to (re)construct the name of a method. The vtables for `vtable` and `symbol` delegate to the vtable for `object`, to ease the creation of singly-rooted hierarchies in which these types are reused directly as end-user object types.

cial about the initial 'vtable vtable' nor the structure of vtables; a new 'vtable vtable' can be created at any time to provide a new `lookup` method that implements a family of vtables with arbitrarily different semantics and internal structure.[5]

3.1 Essential objects and methods

Table 1 lists the three essential object types and the five essential methods that they implement. These methods are described below, with implementations shown in pseudo-code intended to make their operation as clear as possible. (Appendix B presents a complete implementation of these methods and types in GNU C.)

[5] The method `addMethod`, described below, also depends on the internal structure of vtables and would be overridden in parallel with the `lookup` method when changing their structure.

```
let SymbolList = EmptyList

function symbol_intern(self, string) =
  foreach symbol in SymbolList
    if string = symbol.string
      return symbol
  let symbol = new symbol(string)
  append(SymbolList, symbol)
  return symbol
```

Figure 7. Method symbol.intern. Symbols are unique strings. A lazy implementer would co-opt a `vtable` into use as a `SymbolList` holding previously-interned `symbols`.

```
function vtable_addMethod(self, symbol, method) =
  foreach i in 1 .. self.size
    if self.keys[i] = symbol
      self.values[i] := method
      return
  append(self.keys, symbol)
  append(self.values, method)
```

Figure 8. Method vtable.addMethod. If the method name `symbol` is already present, replace the method associated with it. Otherwise add a new association between the name and the method.

Before we can construct an object system we need a way to add methods to `vtables`, which requires a means to construct unique method names. Figure 7 shows a simple algorithm for creating 'interned' (unique) strings that are ideal for use as method names.

To add methods to a `vtable` we send `addMethod` to it, passing a message name (`symbol`) and the address of native code implementing the method. The algorithm is shown in Figure 8.

Sending a message to an object begins by mapping a particular combination of object and message into an appropriate method implementation. Figure 9 shows the algorithm for `vtable`'s `lookup` method that performs this mapping.

Invoking the `allocate` method in a `vtable` allocates a new object. The object is made a member of the vtable's family, as shown in Figure 10.

Finally, the creation of new behaviours is provided by `vtable`'s `delegated` method. It creates a new (empty) `vtable` whose parent is the vtable in which `delegated` was invoked. The algorithm is shown in Figure 11.

```
function vtable_lookup(self, symbol) =
  foreach i in 1 .. self.size
    if self.keys[i] = symbol
      return self.values[i]
  if self.parent ≠nil
    return self.parent.lookup(symbol)
  return nil
```

Figure 9. Method vtable.lookup. The default implementation searches the receiver's keys for the message name. If no match is found the search continues in the parent, if present, otherwise the search fails by answering `nil`.

```
function vtable_allocate(self, size) =
  let object = allocateMemory(PointerSize + size)
  object := object + PointerSize
  object[-1] := self  /* vtable */
  return object
```

Figure 10. Method vtable.allocate. A new object is created and its `vtable` (stored in the word preceding the object) is set to the `vtable` in which the `allocate` method was invoked, making the object a member of that `vtable`'s family. The `size` argument specifies the size of the object's state. Computation of the correct value for `size` is dependent on the programming language implementation in which the object model is being used.

```
function vtable_delegated(self) =
  let child =
    if self ≠nil
      vtable_allocate(self[-1], VtableSize)
    else
      vtable_allocate(nil, VtableSize)
  child.parent := self
  child.keys := EmptyList
  child.value := EmptyList
  return child
```

Figure 11. Method vtable.delegated. A new `vtable` is `allocated` and its `parent` set to the `vtable` in which the `delegated` method is being invoked. These `parent` fields link the `vtables` together into a single delegation chain.

3.2 Message sending

To send a message M to an object O we look up M in the vtable of O to yield a method implementation that is then called. The call passes the object O (which becomes `self` in the called method) and any remaining message arguments. The send algorithm is therefore:

```
function send(object, messageName, args...) =
  let method = bind(object, messageName)
  return method(object, args...)
```

The function bind is responsible for looking up the method name in the vtable of object and just invokes `lookup` in the object's vtable, passing messageName as the argument:

```
function bind(object, messageName) =
  let vt = object[-1]
  let method =
    if messageName = lookup
          and object = VtableVT
      vtable_lookup(vt, lookup)
    else
      send(vt, lookup, messageName)
  return method
```

Note that the recursion implied by `send` calling `bind` which in turn calls `send` (to invoke the `lookup` method in the object's vtable) is broken by 'short-circuiting' the send (calling the method `vtable_lookup` directly) when the method name is `lookup` and the object in which it is being bound is the 'vtable vtable'.

3.3 Bootstrapping the object universe

The structure associated with the three essential types has to be created and their vtables populated before the object model will behave as we have described. Figure 12 shows one possible order in which this initialisation can take place:

1. The vtables for `vtable`, `object` and `symbol` are created and initialised explicitly.
2. The symbol `lookup` is interned and the method `vtable.lookup` installed. At this point the `send` and `bind` functions described in the previous section (i.e., message sending) will work.
3. The symbol `addMethod` is interned and the method `vtable.addMethod` installed. At this point methods can be installed in a vtable by sending `addMethod` to the vtable.
4. The symbol `allocate` is interned and the method `vtable.allocate` installed. At this point new members of an object family can be created by sending their `vtable` the message `allocate`, and this is done to create the prototype `symbol` object.
5. The symbol `intern` is interned and the method `symbol.intern` installed. At this point new symbols can

```
function initialise() =
  /* 1. create and initialise vtables */
  VtableVT := vtable_delegated(nil)
  VtableVT[-1] := VtableVT

  ObjectVT := vtable_delegated(nil)
  ObjectVT[-1] := VtableVT
  VtableVT.parent := ObjectVT

  SymbolVT := vtable_delegated(ObjectVT)

  /* 2. install vtable.lookup */
  lookup := symbol_intern(nil, "lookup")
  vtable_addMethod(VtableVT, lookup, vtable_lookup)

  /* 3. install vtable.addMethod */
  addMethod := symbol_intern(nil, "addMethod")
  vtable_addMethod(VtableVT, addMethod,
                   vtable_addMethod)

  /* 4. install vtable.allocate */
  allocate := symbol_intern(nil, "allocate")
  VtableVT.addMethod(allocate, vtable_allocate)
  symbol := SymbolVT.allocate(SymbolSize)

  /* 5. install symbol.intern */
  intern := symbol.intern(nil, "intern")
  SymbolVT.addMethod(intern, symbol_intern)

  /* 6. install vtable.delegated */
  delegated := symbol.intern("delegated")
  VtableVT.addMethod(delegated, vtable_delegated)
```

Figure 12. Bootstrapping the object model. Method implementations are called as functions and vtable slots initialised explicitly to create the vtables for the three objects types. The methods `symbol_intern` and `vtable_addMethod` are called explicitly to populate the vtables. By the time the last two lines are reached, we have enough of the object model in place that we can send messages to intern the symbol `delegated` and install it in the vtable for vtables.

be interned by sending `intern` to the prototype `symbol` object.

6. Finally, the symbol `delegated` is interned (by sending `intern` to `symbol`) and the method `vtable.delegated` installed (by sending `addMethod` to the vtable for vtables). At this point the object system behaves exactly as described in this paper.

The initialised 'object universe' is shown in Figure 13.

3.4 Implementation language bindings

To deploy the object model as part of a programming language implementation, we need three things:

- implementation language structure definitions for the layouts of `object`, `symbol` and `vtable` (implied by the default implementation of the `lookup` method installed in the 'vtable vtable');
- implementations of the five essential methods in the implementation language; and
- an implementation language method invocation mechanism, to call a method implementation (returned from `lookup`) passing the receiver object and message arguments.

```
#define send(OBJ, MSG, ARGS...) ({                  \
        struct object *o = (struct object *)(OBJ); \
        struct vtable *thisVT = o->_vt[-1];         \
  static struct vtable *prevVT = 0;                 \
  static      method_t  method = 0;                 \
  (thisVT == prevVT                                 \
    ?  method                                       \
    : (prevVT = thisVT,                             \
       method = bind(o, (MSG))))(o, ##ARGS);        \
})
```

Figure 14. Optimising `send` with an inline cache. The `send` macro memoizes the previous vtable and associated closure returned from `bind`. `bind` is only called (and the memoized closure and vtable values updated) if the invocation is to an object whose vtable is not the same as the previous object's vtable at the same invocation site; otherwise the previously bound closure is reused immediately. This is safe provided the method name is a constant at any given invocation site.

Appendix B presents a complete implementation in the GNU C language. The next section discusses two optimisations appearing in this implementation that significantly improve the performance of message sending. It should be straightforward to adapt them to other programming languages.

3.5 Optimising performance

The performance of the GNU C versions of `send()` and `bind()` are improved by two forms of caching.

Figure 14 shows a version of `send` that is implemented as a macro. This allows each send site to remember the previous destination method returned by `bind` in an *inline cache*. As long as the vtable of the next receiver does not change, the previous destination method can be invoked directly without calling `bind` again (assuming the message name at the send site is constant).

Figure 15 shows a version of `bind` that has been optimised with a global *method cache*. Before invoking `lookup` the optimised `bind` looks for the vtable and method name in a cache of previously bound methods. If it finds a match, it returns the cached closure; if not, it invokes `lookup` and fills the appropriate cache line.

These two optimisations are independent and can be used separately or together. Note that a realistic language implementation would need a way to invalidate these caches each time a change is made to vtable contents or inheritance rela-

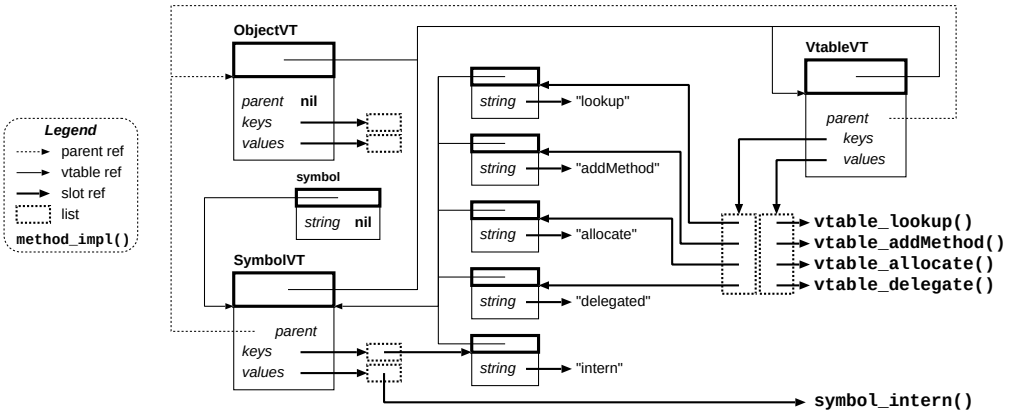

Figure 13. The object model universe. The larger objects are the vtables for the three essential types (`object`, `symbol` and `vtable`). Just above `SymbolVT` is the prototype `symbol` object, and to the right of it are the symbols that provide message names for the five essential methods whose implementations are just below `VtableVT` on the right. The symbol `intern` is bound to the method `string_intern` in the `SymbolVT` and the remaining methods are bound to their message names in `VtableVT`. Both `SymbolVT` and `VtableVT` delegate to `ObjectVT`.

```
struct entry {
  struct vtable  *vtable;
  struct object  *message;
  method_t        method;
} MethodCache[8192];

struct method_t *bind(struct object *obj,
                      struct object *msg)
{
  method_t        m;
  struct vtable  *vt     = obj->_vt[-1];
  unsigned long   offset = hash(vt, msg) & 8191;
  struct entry   *line   = MethodCache + offset;
  if (line->vtable == vt && line->message == msg)
    return line->method;
  m = ((msg == s_lookup) && (obj == vtable_vt))
    ? vtable_lookup(vt, msg)
    : send(vt, s_lookup, msg);
  line->vtable  = vt;
  line->message = msg;
  line->method  = m;
  return m;
}
```

Figure 15. Optimising `bind` with a global method cache. The `MethodCache` stores vtables, message names, and the associated method implementations. To `bind` a message name within a vtable, a hash is computed from the vtable and name modulo the size of the method cache to create a cache line offset. If the vtable and name stored in the cache at that offset correspond to the vtable and name being bound, the stored method is returned immediately. Otherwise `lookup` is invoked in the vtable to bind the method name, and cache updated accordingly.

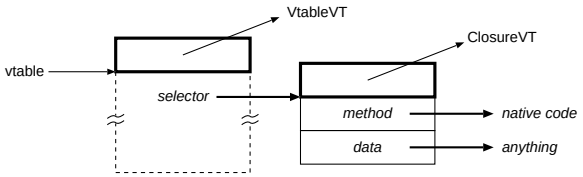

Figure 16. Revised internals of vtables. A vtable maps message names onto closures, containing the address of the native code to be executed and some arbitrary data. Since closures are objects, they too have a pointer to a vtable describing their behaviour.

tionships. Mechanisms for doing this are simple but beyond the scope of this paper.

4. Extensions that improve generality

Section 3 described the simplest possible arrangement of our object model, in which each message name in a vtable is associated with the address of the native code of a corresponding method implementation. We found that the usefulness and generality of the object model were significantly improved by introducing an additional level of indirection, so that a message name is associated with a *closure*. Each closure contains two items: the address of the compiled code implementing the method and some (arbitrary) data, as shown in Figure 16. The `bind` function is modified to return a closure as shown in Figure 17; `send` then invokes the method stored in the closure and passes the closure itself as an argument to the method (in addition to the message receiver and arguments). Method implementations are modified correspondingly, to accept the additional argument.

We believe the slight increase in complexity is more than justified by the generality that is gained. For example:

```
function vtable_addMethod(myClosure, self,
                          aSymbol, aMethod) =
  foreach i in 1 .. self.size
    if self.keys[i] = aSymbol
      self.values[i] := aMethod
      return
  self.keys.append(aSymbol)
  self.values.append(new closure(aMethod, nil))

function send(object, messageName, args...) =
  let closure = bind(object, messageName)
  return closure.method(closure, object, args...)

function bind(object, messageName) =
  let vt = object[-1]
  let closure =
    if messageName = lookup
          and object = VtableVT
      vtable_lookup(nil, vt, lookup)
    else
      send(vt, lookup, messageName)
  return closure
```

Figure 17. Revised methods and functions. The method `addMethod` and the message sending functions `bind` and `send` are modified to store and retrieve closures instead of methods. Note that `addMethod`, like all method implementations, now accepts an additional argument (the closure in which it was found by `lookup`).

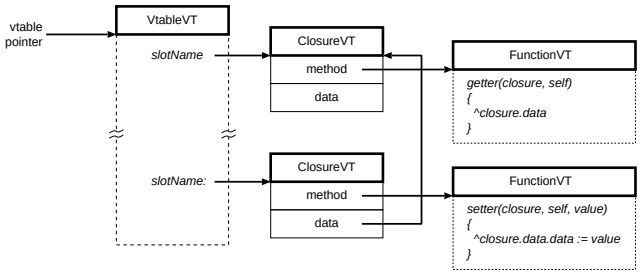

Figure 18. Self-like slots. An assignable slot is implemented as a pair of methods: a 'getter' and a 'setter'. The value of the slot is stored as the `data` in the closure of its getter method. The `data` of the setter method's closure contains a reference to the getter's closure, allowing the setter to assign into the getter's `data`. A single implementation of getter and setter can be shared by all closures associated with assignable slots.

- Figure 18 shows how closures can be used as assignable slots, creating an end-user object model similar to that of traditional prototype-based languages.
- Figure 19 shows how closures are used to support mixed-mode execution [10]. A single interpreter method is shared between many closures whose `data` fields contain the code to be interpreted. To the caller there is no difference between invoking a natively compiled method and invoking an interpreted method.

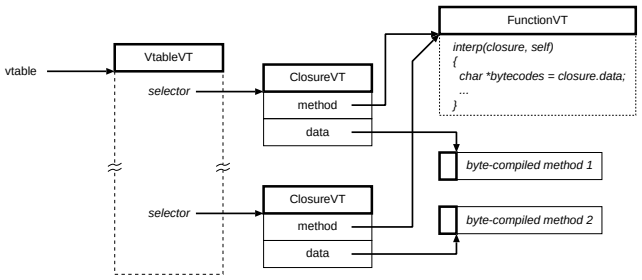

Figure 19. Mixed-mode execution. An interpreter (for bytecodes or other structures) can be shared by any number of method closures. The structure to be interpreted is stored in the `data` part of the closure. As described in the text, the closure is passed as an argument to the method implementation (in this case the interpreter) from where its `data` is readily accessible. To the caller there is no difference between invoking a native method and invoking a byte-compiled method; the calling convention is the same.

Other useful extensions that we have implemented include support for 'Lieberman-style' prototypes [7] which provide much stronger encapsulation than the more common class-based inheritance. A detailed description of this extension is available online [9].

5. Evaluation

We validate our object model in two ways:

- By showing that it can be extended easily to support object models for existing languages or significant and useful features drawn from them. We do this by extending our prototype-based language (that uses our model directly, as described in Section 2.2) first to support the Javascript object model (Section 5.1) and then by adding Traits (Section 5.2).
- By showing that its performance is sufficient for its use in serious language implementations (Section 5.3).

5.1 Ease of use: Javascript objects

Javascript [3] has a simple object model based on delegation [7] in which objects are dictionaries that map property names to their values. When an object is asked for an unknown property, it forwards the request to its prototype (fetched from its `__proto__` property). Properties are 'copy-on-write'; assigning to a property of an object either updates an existing property or creates a new property in the object. All objects, functions and methods in Javascript are based on this model.

Figure 20 shows one way of extending our object model to support these semantics. Note that this implementation is not intended to be used directly by programmers (although

```
vtable get [ ↑closure data ]

get := [ (vtable vtable lookup: #get) method ]

Object set: prop to: val
[
  | closure |
  (closure := self vtable lookup: prop) notNil
    ifFalse:
      [closure := self vtable methodAt: prop put: get].
  closure setData: val.
  prop == #__proto__
    ifTrue:
      [self vtable parent: val vtable.
       vtable flush].
]
```

Figure 20. Javascript objects. Properties are implemented in a manner similar to that of slots in Figure 18. However, setter methods were eliminated in favour of a `set:to:` method that treats the `__proto__` property specially. If `__proto__` is assigned then the `parent` of the object's vtable is set to the `value`'s vtable, and any method caches are flushed. (Note that the block expression assigned to `get` is evaluated; the value assigned is the result of executing the block, not an unevaluated, literal block. Appendix A explains this syntax further.)

nothing prohibits this). Rather, a compiler is expected to translate Javascript expressions into method invocations. For example, a Javascript field access '`x.p`' is translated to '`x p`' (send message `p` to `x`, invoking its property getter). Similarly, the Javascript assignment '`x.p = y`' is translated to '`x set: #p to: y`' (send message `set:to:` to `x`, invoking its property setter) with arguments `#p` (the property name) and `y` (the new value).

5.2 Ease of use: Traits

Traits [11] are a powerful software composition mechanism. A trait is a collection of methods without state that can be manipulated and combined with other traits according to an algebra of composition, aliasing and exclusion. They are interesting because they provide the power of multiple inheritance without the complexity.

Figure 21 shows how our prototype-based language can be extended to support Traits. We can then easily implement the operations of the Traits 'algebra', for example:

```
Trait + aTrait
[
  ↑Trait delegated
    useTrait: self;
    useTrait: aTrait
]
```

This creates a new empty trait and adds both the receiver and the argument to it, composing their behaviours. (Method exclusion and method aliasing are left as an exercise; they take no more than a few minutes each. Once all three operations

```
Trait : Object ()

Object useTrait: aTrait [ aTrait addTo: self ]

Trait addTo: anObject
[
  self vtable keysAndValuesDo: [:selector :closure |
    | newClosure |
    newClosure := anObject vtable
                  traitMethodAt: selector
                  put:           closure method.
    newClosure setData: closure data]
]

vtable traitMethodAt: aSelector put: aMethod
[
  (self includesKey: aSelector)
    ifTrue: [↑self errorConflict: aSelector]
  ↑self methodAt: aSelector put: aMethod
]
```

Figure 21. Support for traits. `Trait.addTo:` adds the methods of the receiver to the vtable of the argument. `vtable.traitMethodAt:put:` adds a method implementation with a given name to the receiver, and signals an error if the method name is already defined.

are available, you will have conforming traits implementation!)

With the above traits implementation in place, we can write code such as:

```
T1 : Trait ()
T1 m  [ 'this is m' putln ]

T2 : Trait ()
T2 n  [ 'this is n' putln ]

C : Object ()  [ C useTrait: T1 + T2 ]

C o   [ self m; n ]
```

(Note that in the above what looks like a literal block after the declaration of `C` is actually an imperative; the program is executed from top to bottom, sending `useTrait:` to `C` before continuing with the installation of method `o` in `C`. Appendix A explains this further.)

5.3 Benchmarks

We measured the size and speed of a sample implementation written in GNU C (see Appendix B), faithfully following the algorithms and structure presented in this paper. All measurements were made on a 2.16 GHz Intel Core Duo.

The sample implementation is approximately 140 lines of code, containing:

- the three essential object types;
- one constructor function, for `symbols`;
- the five essential methods;
- macros for `send` and `bind`, as presented in Section 3.2, with optional inline and global method caches; and

- an initialisation function that creates the initial objects and populates their vtables to create the object system as shown in Figure 13.

The object code size for all essential objects and their methods, with unoptimised `send` and `bind`, is 1,453 bytes. With the inline and global caches enabled, the code size grows to 1,822 bytes.[6] This should not be an issue for any but the most severely resource-constrained environments.

Next we investigate the overhead of dynamic dispatch through the vtables. We implemented the `nfibs` function (which has a very high ratio of message sends, or function invocations, to computation) in optimised C with statically-bound function calls and compared it with our object model using dynamically-bound message sends and an inline cache. The results from running `nfibs(34)` (performing 18,454,929 calls or method invocations) were:

type	*time*	*% of static call*
static call (C)	150 ms	100.0%
dynamic send	270 ms	55.6%

While the results are polluted a little by the arithmetic computation, they show that a static C function call is only approximately twice as fast as a dynamically-bound send through an inline cache. The actual overhead should be lower in practice since most code will perform more computation per call/send than `nfibs`.

Lastly, we implemented the example presented in Section 2 of this paper: data structures suitable for a Lisp-like language. We implemented a 'traditional' `length` primitive using a `switch` on an integer `tag` to select the appropriate implementation amongst a set of possible `case` labels. This was compared with an implementation in which data was stored using our object model and the `length` primitive used `send` to invoke a `method` in the objects themselves.[7] Both were run for one million iterations on forty objects, ten each of the four types that support the `length` operation. The results, with varying degrees of object model optimisations enabled, were:

implementation	*time*	*% of `switch`*
`switch`-based	503 ms	100.0%
dynamic object-based	722 ms	69.7%
+ global cache	557 ms	90.3%
+ inline cache	243 ms	207.0%

This shows that an extensible, object-based implementation can perform at better than half the speed of a typical C implementation for a simple language primitive. With a global method cache (constant overhead, no matter how many method invocation sites exist) the performance is within 10% of optimised C. When the inline cache was enabled the performance was better than twice that of optimised C. In a practical language implementation the above performance gaps would be decrease in all cases as the amount of useful work per primitive increases. (It is hard to conceive of a simpler primitive than `length`.)

5.4 Limitations

Our object model relies on a method cache [2] for performance. It is necessary to flush the cache after certain programming changes such as modifying a vtable (adding or removing a mapping, or storing into the `parent` slot). This is easy to do for both inline and global method caches, but is neither described in this paper nor counted in our evaluation of the sample implementation.

We do not count constructors in the number of methods in the object implementation. (There is no *requirement* for the constructors to be installed as methods although in practice it is convenient to do so.)

We also do not count the vtable pointer as part of the end-user object structure, since it appears before the nominal start of the object.

Lastly, the implementation of `bind` and `send` cannot be exposed as easily as the method lookup mechanism. This can be addressed by exposing the semantics of functions in the same way that the object model exposes the semantics of messaging (see Section 7). This permits almost unlimited flexibility to implement mechanisms such as multimethods.

6. Related work

TinyObjects [6] also lets programmers remove limitations from the system instead of 'programming around' them. It provides a Metaobject Protocol (MOP) [5], at the end-user level of abstraction, that reflects on the implementation level and allows programmers to customise the object model to fit the needs of their applications. We address the same problem by implementing the object model and the equivalent of a very small MOP within a single level of abstraction. This way the programmer can directly manipulate the objects and methods that implement the semantics of their object model.

Smalltalk-80 [4] has methods (in classes Behaviour, Class and Metaclass) that provide what is essentially an incomplete MOP. While these can be used by programs (including the Smalltalk programming environment itself) to create new subclasses and modify method dictionaries, they cannot be used to modify the semantics of message sending itself nor the internal layout of objects.

McCarthy's metacircular evaluator for LISP [8] demonstrated that it is possible for a language to be implemented (described) in itself. Such implementations are 'open': they allow programmers both to write 'user programs' and also to modify or extend the semantics of the language. The circular implementation of our object model brings an equivalent openness to the object-messaging paradigm.

Some systems, such as the Self programming language [12] and Lieberman's prototypes [7], present the user with sim-

[6] Darwin 8.8.1, Intel Core Duo, gcc-4.0.1 (Apple build 5367).

[7] Our reference implementation, including the `length` benchmarks, can be downloaded from: `http://piumarta.com/oopsla07`

pler object models than the one we describe. The cost of this simplicity is that some of the semantics of their object models is hidden (slot lookup in particular) and cannot be modified by end user code. Self also requires a significantly more complex runtime to run efficiently [1]. Our model is much closer Self's internal object model which uses *maps* (similar to our vtables) to describe the behaviour of entire *clone families*. Very promising recent experiments with Self aim to expose the entire implementation to the programmer [13].

7. Conclusions and further work

We presented a simple, extensible object model that exposes its own semantics in terms of the objects and messages that it implements. This circularity in the implementation results in surprising flexibility; end users have direct access to, and control over, the implementation mechanisms of the object model itself. Our experience with this object model has shown that it can be extended easily to support powerful features such as sideways composition and mixed-mode execution. While it is not necessarily a friendly model for hand-written code, it is an attractive target for automatic translation. It could also be an attractive target for statically-typed languages, where the compiler can guarantee runtime type safety.

Because it imposes no structure on end user objects, our model invites experimentation that might otherwise be difficult. For example, it allows a pointer to a compiled native function to also be an object, to which messages can be sent; a vtable in the word before the function prologue suffices. We envisage going further and storing useful information about compiled code (stack layout, signature information, pre- and post-conditions, etc.) in the word before the function's vtable pointer (at offset -2).

This complements our ongoing work with dynamic code generation that brings the functional aspects of our object model (method implementations, method invocation, and `send` and `bind` in particular) under the control of the programmer. This work will be the subject of forthcoming publications.

Starting with the algorithms and C language bindings described in this paper, implementing our object model in C took no more than four hours. The essential objects and methods total 140 lines of source code. Not only is it tiny, but it also scales well: in a slightly different form it has been in daily use by several people for over a year. This model provides rich Smalltalk-like class libraries, implements its own compiler and dynamic code generator for multiple architectures, and integrates seamlessly with platform libraries and garbage collection. With the addition of a few lines of code it can support tagged immediate quantities, and represent the object `nil` with the NULL pointer.

References

[1] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of SELF a dynamically-typed object-oriented language based on prototypes. In *OOPSLA '89: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 49–70, New York, NY, USA, 1989. ACM Press.

[2] L. P. Deutsch and A. M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *POPL '84: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 297–302, New York, NY, USA, 1984. ACM Press.

[3] ECMA. Ecmascript language specification, December 1999. http://www.ecma.ch/ecma1/stand/ecma-262.htm.

[4] A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[5] G. Kiczales, J. des Rivieres, and D. G. Bobrow. *The art of metaobject protocol*. MIT Press, Cambridge, MA, USA, 1991.

[6] G. Kiczales and A. Paepcke. Open Implementations and Metaobject Protocols. `http://www2.parc.com/csl/groups/sda/publications/papers/Kiczales-TUT95/for-web.pdf`.

[7] H. Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *OOPLSA '86: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 214–223, New York, NY, USA, 1986. ACM Press.

[8] J. McCarthy. *LISP 1.5 Programmer's Manual*. The MIT Press, 1962.

[9] I. Piumarta. Efficient Sideways Composition via 'Lieberman' Prototypes. VPRI Research Memo RM-2007-002-a, `http://vpri.org/pdf/prototypes1.pdf`.

[10] S. Sankar, S. Viswanadha, J. H. Solorzano, R. J. Duncan, and D. J. Bacon. Mixed-mode execution for object-oriented programming languages. US Patent 6854113, issued February 8, 2005. `http://www.patentstorm.us/patents/6854113.html`.

[11] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behavior. In *Proceedings ECOOP 2003 (European Conference on Object-Oriented Programming)*, volume 2743 of *LNCS*, pages 248–274. Springer Verlag, July 2003.

[12] D. Ungar and R. B. Smith. Self: The power of simplicity. In *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 227–242, New York, NY, USA, 1987. ACM Press.

[13] D. Ungar, A. Spitz, and A. Ausch. Constructing a metacircular virtual machine in an exploratory programming environment. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 11–20, New York, NY, USA, 2005. ACM Press.

A. Prototype language syntax

The prototype-based language used for several examples in the text has a syntax similar to that of Smalltalk-80 [4] with a few significant differences described here.

A.1 Type declarations

New types are introduced by creating a named prototype of that type. For example,

```
Derived : Base ( a b )
```

creates a variable '`Derived`' (in a kind of 'global namespace') and assigns to it a new prototype belonging to a family of objects that inherit behaviour and state from the family of '`Base`' (another prototype) and which extend that state with two new slots called `a` and `b`. The new vtable for `Derived` is created automatically by sending `delegated` to the vtable for `Base`; this vtable is then sent the message `allocate` to create the prototype stored in `Derived`.

A.2 Method definitions

The body of a method follows its defining message pattern within square brackets. For example,

```
Derived frobble: bob with: bill
[
    ↑bob frobbleWith: bill from: self
]
```

installs the method `frobble:with:` in the vtable for `Derived` by sending it the message `addMethod` with the message name and method implementation as arguments.

A.3 Top-level statements

Arbitrary statements can be executed at the 'top-level' of the program (anywhere a definition is allowed) by enclosing them in square brackets. For example,

```
[
    'running DeepThought program...' putln.
    DeepThought new multiply: 6 by: 9.
]
```

announces to the user that an application is about to run, then instantiates and runs it.

A.4 Top-level definitions

Variables in the 'global namespace' can be bound to arbitrary values (not just to new prototypes as described above). For example,

```
TheAnswer := [ 42 ]
```

creates a 'global' variable named `TheAnswer` and initialises it with the value of the last expression in the block (in this case, the literal `42`).

B. Sample object model implementation

```
/*  A sample implementation in GNU C of the object model described in this paper.
 *  This code, and that of the benchmarks discussed in the text, can be downloaded from: http://piumarta.com/oopsla07
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define ICACHE 1        /* nonzero to enable point-of-send inline cache */
#define MCACHE 1        /* nonzero to enable global method cache        */

struct vtable;
struct object;
struct symbol;

typedef struct object *(*method_t)(struct object *receiver, ...);

struct vtable
{
  struct vtable  *_vt[0];
  int             size;
  int             tally;
  struct object **keys;
  struct object **values;
  struct vtable  *parent;
};

struct object {
  struct vtable *_vt[0];
};

struct symbol
{
  struct vtable *_vt[0];
  char          *string;
};

struct vtable *vtable_vt   = 0;
struct vtable *object_vt   = 0;
struct vtable *symbol_vt   = 0;

struct object *s_addMethod = 0;
struct object *s_allocate  = 0;
struct object *s_delegated = 0;
struct object *s_lookup    = 0;
struct object *s_intern    = 0;

struct object *symbol      = 0;

struct vtable *SymbolList  = 0;

extern inline void *alloc(size_t size)
{
  struct vtable **ppvt= (struct vtable **)calloc(1, sizeof(struct vtable *) + size);
  return (void *)(ppvt + 1);
}

struct object *symbol_new(char *string)
{
  struct symbol *symbol = (struct symbol *)alloc(sizeof(struct symbol));
  symbol->_vt[-1] = symbol_vt;
  symbol->string = strdup(string);
  return (struct object *)symbol;
}

struct object *vtable_lookup(struct vtable *self, struct object *key);

#if ICACHE
# define send(RCV, MSG, ARGS...) ({                           \
             struct object *r = (struct object *)(RCV);       \
             struct vtable *thisVT = r->_vt[-1];              \
      static struct vtable *prevVT = 0;                       \
      static      method_t  method = 0;                       \
      (thisVT == prevVT                                       \
        ? method                                              \
        : (prevVT = thisVT,                                   \
           method = _bind(r, (MSG))))(r, ##ARGS);             \
    })
#else
# define send(RCV, MSG, ARGS...) ({                           \
```

```
      struct object *r      = (struct object *)(RCV);           \
      method_t       method = _bind(r, (MSG));                  \
      method(r, ##ARGS);                                        \
    })
#endif

#if MCACHE
struct entry {
  struct vtable  *vtable;
  struct object  *selector;
  method_t        method;
} MethodCache[8192];
#endif

method_t _bind(struct object *rcv, struct object *msg)
{
  method_t       method;
  struct vtable *vt = rcv->_vt[-1];
#if MCACHE
  unsigned int   hash = (((unsigned)vt << 2) ^ ((unsigned)msg >> 3)) & ((sizeof(MethodCache) / sizeof(struct entry)) - 1);
  struct entry  *line = MethodCache + hash;
  if (line->vtable == vt && line->selector == msg)
    return line->method;
#endif
  method = ((msg == s_lookup) && (rcv == (struct object *)vtable_vt))
    ? (method_t)vtable_lookup(vt, msg)
    : (method_t)send(vt, s_lookup, msg);
#if MCACHE
  line->vtable   = vt;
  line->selector = msg;
  line->method   = method;
#endif
  return method;
}

struct object *vtable_allocate(struct vtable *self, int payloadSize)
{
  struct object *object = (struct object *)alloc(payloadSize);
  object->_vt[-1] = self;
  return object;
}

struct vtable *vtable_delegated(struct vtable *self)
{
  struct vtable *child= (struct vtable *)vtable_allocate(self, sizeof(struct vtable));
  child->_vt[-1] = self ? self->_vt[-1] : 0;
  child->size    = 2;
  child->tally   = 0;
  child->keys    = (struct object **)calloc(child->size, sizeof(struct object *));
  child->values  = (struct object **)calloc(child->size, sizeof(struct object *));
  child->parent  = self;
  return child;
}

struct object *vtable_addMethod(struct vtable *self, struct object *key, struct object *method)
{
  int i;
  for (i = 0;  i < self->tally;  ++i)
    if (key == self->keys[i])
      return self->values[i] = (struct object *)method;
  if (self->tally == self->size)
    {
      self->size  *= 2;
      self->keys   = (struct object **)realloc(self->keys,   sizeof(struct object *) * self->size);
      self->values = (struct object **)realloc(self->values, sizeof(struct object *) * self->size);
    }
  self->keys  [self->tally  ] = key;
  self->values[self->tally++] = method;
  return method;
}

struct object *vtable_lookup(struct vtable *self, struct object *key)
{
  int i;
  for (i = 0;  i < self->tally;  ++i)
    if (key == self->keys[i])
      return self->values[i];
  if (self->parent)
    return send(self->parent, s_lookup, key);
  fprintf(stderr, "lookup failed %p %s\n", self, ((struct symbol *)key)->string);
  return 0;
```

```
}

struct object *symbol_intern(struct object *self, char *string)
{
  struct object *symbol;
  int i;
  for (i = 0;  i < SymbolList->tally;  ++i)
    {
      symbol = SymbolList->keys[i];
      if (!strcmp(string, ((struct symbol *)symbol)->string))
        return symbol;
    }
  symbol = symbol_new(string);
  vtable_addMethod(SymbolList, symbol, 0);
  return symbol;
}

#define trace() printf("%s %d\n", __FUNCTION__, __LINE__);  fflush(stdout)

void init(void)
{
  vtable_vt = vtable_delegated(0);
  vtable_vt->_vt[-1] = vtable_vt;

  object_vt = vtable_delegated(0);
  object_vt->_vt[-1] = vtable_vt;
  vtable_vt->parent = object_vt;

  symbol_vt = vtable_delegated(object_vt);
  SymbolList = vtable_delegated(0);

  s_lookup = symbol_intern(0, "lookup");
  vtable_addMethod(vtable_vt, s_lookup,    (struct object *)vtable_lookup);

  s_addMethod = symbol_intern(0, "addMethod");
  vtable_addMethod(vtable_vt, s_addMethod, (struct object *)vtable_addMethod);

  s_allocate = symbol_intern(0, "allocate");
  send(vtable_vt, s_addMethod, s_allocate,  vtable_allocate);
  symbol = send(symbol_vt, s_allocate, sizeof(struct symbol));

  s_intern = symbol_intern(0, "intern");
  send(symbol_vt, s_addMethod, s_intern, symbol_intern);

  s_delegated = send(symbol, s_intern, (struct object *)"delegated");
  send(vtable_vt, s_addMethod, s_delegated, vtable_delegated);
}
```