Open, reusable object models

Ian Piumarta
Viewpoints Research Institute
9242 Beverly Blvd, Suite 300
Beverly Hills, CA 90210, USA
piumarta@speakeasy.net

Alessandro Warth
UCLA Computer Science Department
3400 Boelter Hall
Los Angeles, CA 90095, USA
awarth@cs.ucla.edu

Abstract

Code reuse between different object model implementations is rare. Most object models cannot easily be shared because they are implemented at a lower level of abstraction than that of the language in which they are designed to operate, rendering their semantics opaque and unavailable for modification by end users. We show that three object types and four methods are sufficient to bootstrap an object model whose semantics are described entirely in terms of those same objects and messages. The result is a simple but powerful object model that can be implemented easily and efficiently, in which all semantics are exposed and malleable, facilitating extensibility, interoperability and implementation reuse.

Categories and Subject Descriptors D.1.5 [*Programming Techniques*]: Object-oriented Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features—Classes and Objects

General Terms Programming Language Design, Programming Language Implementation

Keywords object model, open implementation

1. Introduction

There are almost as many ways to represent structured data as there are programming languages to manipulate that data. Once a representation has been chosen for a particular *end user language* it is typically described in some lower-level *implementation language*. To illustrate this, the implementer of a Lisp-like end user language might choose C as the implementation language and use a *discriminated union* to store structured data:

```
enum RecordTag { Number, Symbol, Cons };

struct Record {
  enum RecordTag  tag;
  union {
    struct Number number;
    struct Symbol symbol;
    struct Cons   cons;
  } payload;
};
```

Each primitive in the end user language that manipulates structured data would then use conditional `if` or `switch` statements to select appropriate behaviour depending on the value of 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 three 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 modification in the future.

These problems are more severe when we consider object-oriented languages. The object model for a simple prototype-based language might specify a `methodDictionary` and a `parent` field 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 (and so on until reaching the end of the parent chain). Adding multiple inheritance to this language would be very difficult because of the runtime's assumption that the `parent` field contains a single object rather than, say, a list of parent objects to try in turn.

The trouble is that some of the semantics are reified eagerly in the execution mechanisms of the language. This in turn eagerly imposes supporting meta-structure 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 customised by end users. Moreover, adapting the model for use in a different language is more difficult when the required changes are pervasive and expressed in a low-level implementation language. In this paper we construct an object model that eliminates all of these problems to the greatest extent possible:

- 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 (possibly *the* simplest to implement) extensible object model that imposes no structure at all on end user objects (Section 3).
- This object model implements its own message-passing semantics in terms of the objects that it implements, making the implementation replaceable from within the end user language. We show that five small methods and four initial objects are sufficient to achieve this (Section 3.1).
- The flexibility gained by exposing the object model's semantics is illustrated by showing how it can be extended easily to support language features such as multiple inheritance and mixed-mode execution (Sections 2.2 and 3).
- We validate the use of this approach for production systems by showing that: it has low space overhead (Section 4); its performance can be competitive with, or even better than, equivalent 'static' implementation techniques (Section 4.1); that an existing object system can be easily implemented on top of our model (Section 4.2); and that advanced compositional techniques such as traits can be accommodated (Section 4.3).

2. The problem

This section sets the stage for our object model by pursuing the examples mentioned in the introduction.

2.1 Adding data types to a language

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

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

Let's add a vector type to this language. We have to extend the `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. However, it is much worse if we want to share our new type with other users of the language, maybe as a third-party extension.

It would be better to associate the `case`s within the primitive function with the data types themselves. Using our object model the new data type is added to the language by creating a new kind of object behaviour (called a *vtable*) and then installing its primitive functionality in the vtable. Figure 1 shows what the additions would look like.

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 statically at compile time; adding a new virtual method would invalidate all previous assumptions about the vtable layout.

```
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 `s_addMethod` method 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.)

Perhaps more compelling is an example involving an object-oriented language that uses our object model directly and that can consequently directly modify the implementation of its own object model.

2.2 Converting single inheritance to multiple inheritance

We can easily create a prototype-based high-level programming language that uses our object model directly for its end user objects.[1] To modify the semantics of message sending in this language we need only replace the mechanism that finds a method implementation given an object and the name of a method to invoke.

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 the corresponding method implementation stored in the vtable. By redefining this method we can change the semantics of method lookup.

Our prototype-based language provides the programmer with single inheritance: a given family of objects can inherit behaviour from a parent family, and all families eventually inherit 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  [ StdOut nextPutAll: 'this is m'; cr ]

C2 : Object()
C2 n  [ StdOut nextPutAll: 'this is n'; cr ]
```

[1] This language is written entirely in itself and can be downloaded, along with many example programs, from `http://piumarta.com/pepsi`

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

```
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: methodName
[
  | method |
  self do: [:vt |
    (method := vt lookup: methodName) 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 `methodName` in each parent consecutively. (The `lookup:` method already installed in `vtable` can be left in place; it searches up the inheritance chain by invoking `lookup:` in its `parent` slot, which can now be either a `vtable` or a `ParentList`.)

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

the programmer can now dynamically add `C2` as a parent of `C1`:

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

such 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 into account, although this could be as simple as only allowing one parent to be stateful (the others effectively being *traits*).

3. Open 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 quantity in which a method can be invoked by name; 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. 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.

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.

A vtable is an object too, and has a reference to a 'vtable for vtables' before its internal state. The 'vtable for vtables' is its own vtable as shown in Figure 5. The state within a vtable describes

[3] The discussion of related work (Section 5) mentions Self, a system that made the opposite choice of modeling behaviour as a special kind of state.

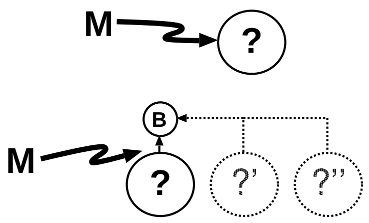

Figure 3. Minimal object model. An object is some unknown state *?* on which a method M can be invoked by name. To implement this model we need a mapping from method names to method implementations. So, to invoke a method M' in the object *?'* we find the corresponding method implementation in a behaviour description B'. Hence an object is 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 *?'*, *?''*, *?'''*, ...

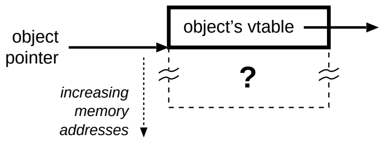

Figure 4. Implementation of minimal object. An object pointer (oop) points to the start of the object's internal state (if any). The object's behaviour is described by a *virtual table* (vtable). A pointer to the vtable is placed one word before the object's state.

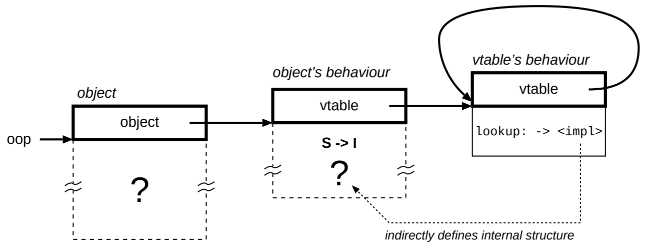

Figure 5. Everything is an object. Every object has a vtable that describes its behaviour. A method is looked up in a vtable by invoking its `lookup:` method. Hence there is a 'vtable vtable' that provides an implementation of `lookup:` for all vtables in the system, including for itself. The implementation of this `lookup:` method is the only thing in the object model that imposes internal structure on vtables. (If the figure seems confusing, try thinking of the word in the solid box as the 'type' of the object to which it is attached.)

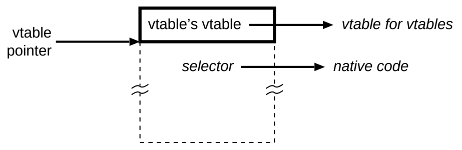

Figure 6. Internals of vtables. A vtable maps method names (selectors) onto method implementations. A method implementation is represented as a *closure* containing the address of the native code to be executed along with some arbitrary data. Since vtables and closures are objects, they each have a vtable pointer in the word before their state.

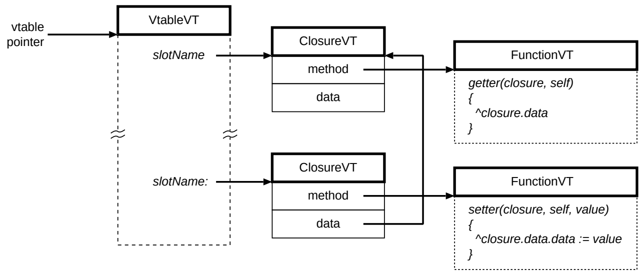

Figure 7. Self-like slots. An assignable slot is implemented as a pair of methods: a 'getter' and a 'setter'. The value of the slot is stored as the `data` in the closure of its getter method. The `data` of the setter method's closure contains a reference to the getter's closure, allowing the setter to assign into the getter's `data`. A single implementation of getter and setter can be shared by all closures associated with assignable slots.

the mapping between method names and the corresponding method implementations. The 'vtable vtable' defines the method `lookup:` for all vtables, used to find a method implementation associated with a method name. This `lookup:` method indirectly defines the internal structure of all vtables, but there is nothing special about the initial 'vtable vtable' nor the structure of vtables: a new 'vtable vtable' can be created at any time and given a `lookup:` method that implements a family of vtables with arbitrarily different internal structure and semantics.

The simplest possible arrangement would be for each selector in the vtable to be associated with the address of the compiled native code that implements the method. We chose to introduce an additional level of indirection, so that a selector is instead associated with a *closure*. Each closure then contains two items: the address of the compiled code implementing the method and some (arbitrary) data.

The final arrangement of our vtables is shown in Figure 6. We believe the slight increase in complexity is more than justified by the generality that is gained. For example:

- Figure 7 illustrates closures used to store assignable slots, creating an end user object model similar to that of traditional prototype-based languages.
- Figure 8 shows how closures are used to support mixed-mode execution. 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 or a byte-compiled method.

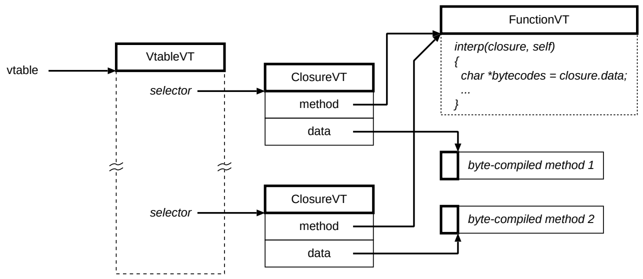

Figure 8. Mixed-mode execution. An interpreter (for bytecodes or other structures) can be shared by any number of method closures. The structure to be interpreted is stored in the `data` part of the closure. As described in the text, the closure is passed as an argument to the method implementation (in this case the interpreter) from where its `data` is readily accessible. To the caller there is no difference between invoking a native method and a byte-compiled method; the calling conventions are identical.

prototype	*method*
vtable	addMethod
vtable	lookup
vtable	allocate
symbol	intern
closure	*constructor*
vtable	delegated

Table 1. Essential objects and methods. The most important methods are applicable to `vtables`: `addMethod` constructs vtables by adding an association from a method name to an implementation, `lookup` queries the associations to find an implementation corresponding to a name, and `allocate` creates a new object (whose `vtable` is the object in which the method was invoked). We include `symbol`'s `intern` method since the end user must have some way to construct the name of a method. The `closure` prototype is required (for cloning when adding methods to a vtable) but has no required methods defined for it. A delegation mechanism is not required, but is trivial to implement in the intrinsic object model; in that case a `delegated` method for `vtable` is required, to create a new `vtable` that delegates to the one in which `delegated` was invoked.

3.1 Essential objects and methods

Table 1 lists the three essential object types and the four essential methods defined for them.[4]

Before we can begin constructing an object system we need a way to add methods to `vtables`, which in turn means we need a way to construct unique method names. Figure 9 shows a simple algorithm for creating 'interned' (unique) strings that are perfect for use as method names.

To add methods to a `vtable` we invoke its `addMethod` method, passing a message name `symbol` and the address of native code implementing the method. The algorithm is shown in Figure 10.

To invoke a method it must first be looked up in a `vtable`. Figure 11 shows the algorithm for `vtable`'s `lookup` method.

[4] Note that constructors (other than the general end user object constructor `allocate`) are not considered methods.

```
let SymbolList = EmptyList
method symbol.intern(string) =
  foreach symbol in SymbolList
    if string = symbol.string
      return symbol
  let symbol = new symbol(string)
  SymbolList.append(symbol)
  return symbol
```

Figure 9. Method symbol.intern. Symbols are unique strings. A lazy implementation could co-opt a `vtable` into use as the `SymbolList`.

```
method vtable.addMethod(symbol, method) =
  foreach i in 1 .. self.size
    if self.keys[i] = symbol
      self.values[i] := method
      return
  self.keys.append(symbol)
  self.values.append(new closure(method, nil))
```

Figure 10. 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.

```
method vtable.lookup(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 11. Method vtable.lookup. The two lines involving `parent` are not essential, but are trivial to include and provide single delegation as part of the intrinsic object model.

```
method vtable.allocate(size) =
  let object = allocateMemory(PointerSize + size)
  let object = object + PointerSize
  object[-1] := self  /* vtable */
  return object
```

Figure 12. Method vtable.allocate. A new object is created and its `vtable` (stored in the word preceding the object) set to the `vtable` in which the method is invoked. 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.

```
method vtable.delegated() =
  let child = self.allocate(VtableSize)
  child.parent := self
  return child
```

Figure 13. 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 `vtable`s together into a single delegation chain.

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 12.

If single delegation is being included in the intrinsic object model, an additional method can be installed in `vtable`: `delegated` creates a new `vtable` whose parent is the vtable in which `delegated` was invoked. The algorithm as shown in Figure 13 is far easier to understand than this verbose description.

3.2 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 `symbol`, `closure` and the `vtable` implied by the `lookup` method of the 'vtable vtable';
- implementations of the four essential methods in the implementation language; and
- implementation language *bindings* to method invocation (the process that invokes the result of `lookup` in an object).

By now it should be clear how to construct the first two. To illustrate the third, we will create bindings to our object model for the GNU C language.

To invoke a method named M in an object O we look up M in the vtable of O to yield a closure, and then call the native method stored in the closure. This call passes the closure, the object O (which becomes `self` in the invoked method) and any remaining arguments in the invocation. If we call this 'sending a message' to M then the function send can be defined as:

```
function send(object, messageName, args...) =
  let closure = bind(object, messageName)
  return closure.method(closure, 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 methodName as the argument:

```
function bind(object, messageName) =
  let vtable = object[-1]
  if messageName = SymbolLookup
        and object = VtableVtable
    let closure = vtable_lookup(0, vtable, messageName)
  else
    let closure = send(vt, SymbolLookup, messageName)
  return closure
```

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 to the intrinsic definition of `vtable_lookup` when the method name is `lookup` and the object in which it is being bound is the 'vtable vtable'. This can be seen both in the above algorithm and in the working translations to C of `send` and `bind` in Figure 14.

```
struct object { struct _vt[0]; };
struct vtable { struct _vt[0]; ...internals... };

#define send(OBJ, MSG, ARGS...) ({               \
  struct object  *o = (struct object *)(OBJ); \
  struct closure *c = bind(o, (MSG));         \
  c->method(c, o, ##ARGS);                    \
})

struct closure *bind(struct object *obj,
                     struct object *msg)
{
  struct vtable *vt = obj->_vt[-1];
  return (    (msg == s_lookup)
          && (obj == (struct object *)vtable_vt))
    ? (struct closure *)vtable_lookup(0, vt, msg)
    : (struct closure *)send(vt, s_lookup, msg);
}
```

Figure 14. Simple method invocation. The `_vt` member of the `structs` is empty but gives access to the vtable pointer stored at offset -1. The `send` operation is implemented as a macro and inlined at each send site. It evaluates the `OBJ` expression once and calls `bind` on the result to find a `closure`. The method stored in the closure is invoked passing the closure, the object, and any remaining arguments. The `bind` function extracts the vtable from the object and invokes `lookup` in it. If the method name being bound is `lookup` and the object in which it is being invoked is the 'vtable vtable' then the lookup is short-circuited to its native method `vtable_lookup` (creating a fixed point in the recursive definition of method lookup); otherwise a full `send` is performed to lookup and then invoke the `lookup` method in the object's vtable, to find a method implementation for the name being bound.

We always pass the bound closure as the first argument of the invoked native method. This gives shared methods access to any differentiating `data` that might be stored in the closure.

3.3 Bootstrapping the object universe

A small amount of structure has to be created before the object model will behave as we have described:

1. the symbols associated with the essential method names must be constructed; then
2. the 'vtable vtable' is constructed and its circular `vtable` reference pointed back to itself; then
3. the methods `vtable.lookup` and `vtable.addMethod` can be installed by calling the native `vtable_addMethod` method directly; we can now invoke `addMethod` in `vtable` using the `send` macro, so
4. the method `vtable.allocate` is added to `vtable` by sending `addMethod` with appropriate arguments; then
5. vtables for `symbol` and `closure` can be allocated.

Figure 15 shows this process for a C implementation of the object model as it appears in our reference implementation (including the optional support for single delegation).

3.4 Optimising performance

The performance of both `send` and `bind` can be improved by caching.

Figure 16 shows a version of `send` that caches the previous vtable and closure resulting from binding the method name in the vtable. Provided the message name is a constant (does not change

```
void initialise(void)
{
  SymbolList = vtable_delegated(0, 0);

  s_lookup    = symbol_intern(0, 0, "lookup");
  s_addMethod = symbol_intern(0, 0, "addMethod");
  s_allocate  = symbol_intern(0, 0, "allocate");
  s_delegated = symbol_intern(0, 0, "delegated");

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

  vtable_addMethod(0, vtable_vt,
                   s_lookup,    (imp_t)vtable_lookup);
  vtable_addMethod(0, vtable_vt,
                   s_addMethod, (imp_t)vtable_addMethod);

  send(vtable_vt, s_addMethod,
       s_allocate,  vtable_allocate);
  send(vtable_vt, s_addMethod,
       s_delegated, vtable_delegated);

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

  symbol_vt  = vtable_delegated(0, object_vt);
  closure_vt = vtable_delegated(0, object_vt);
}
```

Figure 15. Initialising the object model. A `vtable` is pressed into service as a `SymbolList`. Variables that begin with `s_` are method names (symbols). Variables that end with `_vt` are the vtables for the various kinds of object. Functions called `x_y` are the method implementations destined to be installed as method `y` in the vtable for objects of type `x`. Whenever a method implementations is called directly a `0` is passed as the first argument, corresponding to the `closure` that would be passed implicitly if we were to use `send` instead.

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

Figure 16. Optimising `send` with an inline cache. The macro `send` 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.

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

struct closure *bind(struct object *obj,
                     struct object *msg)
{
  struct closure *c;
  struct vtable  *vt = obj->_vt[-1];
  struct entry   *cl = MethodCache
    + ((((unsigned)vt  << 2) ^ ((unsigned)msg >> 3))
       & ((sizeof(MethodCache) / sizeof(struct entry))
           - 1));
  if (cl->vtable == vt && cl->message == msg)
    return cl->closure;
  c = (   (msg == s_lookup)
       && (obj == (struct object *)vtable_vt))
    ? (struct closure *)vtable_lookup(0, vt, msg)
    : (struct closure *)send(vt, s_lookup, msg);
  cl->vtable  = vt;
  cl->message = msg;
  cl->closure = c;
  return c;
}
```

Figure 17. Optimising `bind` with a global method cache. The `MethodCache` stores vtables, method names, and the associated closures. To `bind` a method name within a vtable, a hash is computed from the vtable and name modulo the size of the method cache to create an index. If the vtable and name stored in the cache at that index correspond to the vtable and name being bound, the stored closure is returned immediately. Otherwise `lookup` is invoked in the vtable to bind the method name, and cache updated accordingly.

between consecutive sends) the cached closure can be reused without calling `bind`.[5]

Figure 17 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.[6]

These two optimisations are independent and can be used separately or together.

4. Evaluation

We measured speed and size, then tried to estimate ease of use objectively. All measurements were made on a 2.16 GHz Intel Core Duo.

Our sample implementation (faithfully following the algorithms and C language bindings presented in this paper, and including optional single delegation) is 144 lines of code and contains:

- the three essential object types, and a negligible common parent type `object`;
- two constructors, for `symbol` and `closure`;
- the four essential methods plus `vtable.delegated`;
- `send` as presented, with defeatable inline cache;
- `bind` as presented, with defeatable global method cache;
- an initialisation method that creates the initial objects and populates their vtables.

The object code size for all essential objects and their methods, with unoptimised `send` and `bind`, is 1,451 bytes. With the inline and global caches enabled, the code size grows to 1,602 bytes. This should not be an issue for any but the most severely resource-constrained environments.

4.1 Benchmarks

The first thing we have to investigate is the overhead of dynamic dispatch through the vtable.

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

type	*time*	*% C*
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`.

Next 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 called `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*	*% C*
`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 is 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`.)

4.2 Ease of use: Javascript objects

Javascript [3] has a simple object model based on delegation [7]. 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 (the value of its `__proto__` property). (Assigning to a property of an object always updates an existing or

[5] A realistic language implementation would need a way to invalidate the inline caches. Mechanisms for doing this are beyond the scope of this paper.

[6] A realistic language implementation would have to invalidate all or parts of the global method cache on every change to vtable contents or inheritance relationships.

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

```
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 18. Javascript objects. Properties are implemented in a manner similar to that of slots in Figure 7. 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.)

creates a new property in the object.) This simple model really is sufficient to implement all Javascript objects, functions and methods.

Figure 18 shows one way of implementing these semantics in our object model. This implementation is small, runs efficiently, can be understood easily, and took very little time to construct.[8]

We should stress that this implementation is not intended to be used directly by programmers (although nothing prohibits this). Rather, a compiler is expected to translate Javascript expressions into method invocations. For example, a field access `x.p` is translated to `x p` (an invocation of method `p` in `x`). Similarly, the assignment `x.p = y` is translated to `x set: #p to: y` (an invocation of `set:to:` in `x` with arguments `#p` (the name `p`) and `y`).

4.3 Ease of use: traits

Traits [9] 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 drawbacks and complexity.

Figure 19 shows support for traits added to our prototype-based language. 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 are available, you will have a genuine, conformant traits implementation.)

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

[8] Admittedly, the person who constructed it was already an expert in the underlying object model.

```
Trait : Object ()

Object useTrait: aTrait [ aTrait addTo: self ]

Trait addTo: anObject
[
  self vtable keysAndValuesDo: [:key :value |
    | 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 19. Support for traits. `Trait.addTo:` adds the methods of the receiver to the vtable of the argument. `vtabel.traitMethodAt:put:` adds a method implementation to the receiver with a given name, and signals an error if the method name is already defined.

```
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`.)

4.4 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 adding a new method to a vtable or storing into a vtable's `parent`, etc.

We don't 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 don't count the vtable pointer as part of the end-user object structure, since it appears before the nominal start of the object. This might be considered dishonest.

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 pure functions (Section 6) in the same way that the object model exposes the semantics of lookup.

5. Related work

TinyObjects [6] is an impressive attempt to simplify applications by empowering the programmer to remove limitations from the system instead of 'programming around' them. It provides a Metaobject Protocol (MOP) [5] through which programmers can customise the object model to fit the needs of their applications. By the authors' own admission, when a particular customisation cannot be expressed with the MOP the programmer must contact the MOP designer and request that the MOP be extended with new functionality. This lack of control over the language is exactly the problem MOPs were designed to solve in the first place. Instead of solving the problem MOPs simply move it to a different (the 'meta') level. We address this problem by implementing the object model and the equivalent of a MOP at the same level of abstraction, giving the programmer control over all aspects of the object model implementation.[9]

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 object models, such as those of the Self programming language [10] and Lieberman's prototypes [7], are simpler than the one we describe. The cost of this simplicity is that some of the semantics of their object model is hidden (slot lookup in particular) and cannot be modified from end user code. Self also requires a significantly more complex runtime to run efficiently. 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* [1]. Very promising recent experiments with Self aim to expose the entire implementation to the programmer [11].

6. 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 customised easily to support powerful features such as multiple inheritance and mixed-mode execution. While it is not necessarily a friendly model for hand-written code, it is an attractive target for automatic translation.

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.

This complements ongoing work with dynamic code generation that brings the purely functions aspects of our object model (method implementations and 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 144 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 also supports tagged immediate quantities and the object `nil` represented as the NULL pointer.

[9] The authors of TinyObjects state that MOPs are expected to stabilise and eventually accommodate "most of the substrate adjustments that are reasonable". We think 'reasonable' is in the eye of the programmer, not the MOP designer. Nobody can know (or predict) which parts of their systems programmers will need to modify in the future. The only way to implement the future is to avoid having to predict it.

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] 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.

[10] 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.

[11] 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. Example object model implementation

An example implementation in C of the object model described in this paper.[10]

```
#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 closure;
struct symbol;

typedef struct object *(*imp_t)(struct closure *closure, 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 closure
{
  struct vtable *_vt[0];
  imp_t          method;
  struct object *data;
};

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

struct vtable *SymbolList= 0;

struct vtable *vtable_vt;
struct vtable *object_vt;
struct vtable *symbol_vt;
struct vtable *closure_vt;

struct object *s_addMethod = 0;
struct object *s_allocate  = 0;
struct object *s_delegated = 0;
struct object *s_lookup    = 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 *closure_new(imp_t method, struct object *data)
{
  struct closure *closure = (struct closure *)alloc(sizeof(struct closure));
  closure->_vt[-1] = closure_vt;
  closure->method  = method;
  closure->data    = data;
  return (struct object *)closure;
}
```

[10] This code and that of the two benchmarks in the following sections can be downloaded from: `http://piumarta.com/pepsi/objmodel.tar.gz`

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

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

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

struct closure *bind(struct object *rcv, struct object *msg)
{
  struct closure *c;
  struct vtable  *vt = rcv->_vt[-1];
#if MCACHE
  struct entry   *cl = MethodCache + ((((unsigned)vt << 2) ^ ((unsigned)msg >> 3))
                                      & ((sizeof(MethodCache) / sizeof(struct entry)) - 1));
  if (cl->vtable == vt && cl->selector == msg)
    return cl->closure;
#endif
  c = ((msg == s_lookup) && (rcv == (struct object *)vtable_vt))
    ? (struct closure *)vtable_lookup(0, vt, msg)
    : (struct closure *)send(vt, s_lookup, msg);
#if MCACHE
  cl->vtable   = vt;
  cl->selector = msg;
  cl->closure  = c;
#endif
  return c;
}

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

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

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

struct object *symbol_intern(struct closure *closure, 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(0, SymbolList, symbol, 0);
  return symbol;
}

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

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

  symbol_vt  = vtable_delegated(0, object_vt);
  closure_vt = vtable_delegated(0, object_vt);

  SymbolList = vtable_delegated(0, 0);

  s_lookup    = symbol_intern(0, 0, "lookup");
  s_addMethod = symbol_intern(0, 0, "addMethod");
  s_allocate  = symbol_intern(0, 0, "allocate");
  s_delegated = symbol_intern(0, 0, "delegated");

  vtable_addMethod(0, vtable_vt, s_lookup,    (imp_t)vtable_lookup);
  vtable_addMethod(0, vtable_vt, s_addMethod, (imp_t)vtable_addMethod);

  send(vtable_vt, s_addMethod, s_allocate,    vtable_allocate);
  send(vtable_vt, s_addMethod, s_delegated,   vtable_delegated);
}
```

B. Object model benchmark

The object model benchmark discussed in the body of the paper. It uses the example object model implementation in C from the previous section to provide a few data types and primitives that discriminate between them using dynamic dispatch.

```
struct object *s_new= 0;
struct object *s_length= 0;

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

struct vtable *Number_vt = 0;
struct object *Number = 0;

struct object *Number_new(struct closure *closure, struct Number *self)
{
  fprintf(stderr, "Number_new\n");
  exit(1);
  return 0;
}

int Number_length(struct closure *closure, struct Number *self)
{
  fprintf(stderr, "Number has no length\n");
  exit(1);
  return 0;
}

struct String
{
  struct vtable *_vt[0];
  int            length;
  char          *chars;
};

struct vtable *String_vt = 0;
struct object *String = 0;

struct object *String_new(struct closure *closure, struct String *self, int size)
{
  struct String *clone = (struct String *)send(self->_vt[-1], s_allocate, sizeof(struct String));
  clone->length = size;
  clone->chars  = (char *)malloc(size);
  return (struct object *)clone;
}

int String_length(struct closure *closure, struct String *self)
{
  return self->length;
}

struct Symbol
{
  struct vtable *_vt[0];
  struct String *string;
};

struct vtable *Symbol_vt = 0;
struct object *Symbol = 0;

struct object *Symbol_new(struct closure *closure, struct Symbol *self, struct String *string)
{
  struct Symbol *clone = (struct Symbol *)send(self->_vt[-1], s_allocate, sizeof(struct Symbol));
  clone->string = string;
  return (struct object *)clone;
}

int Symbol_length(struct closure *closure, struct Symbol *self)
{
  return self->string->length;
```

```
}

struct Vector
{
  struct vtable *_vt[0];
  int            length;
  struct object *contents;
};

struct vtable *Vector_vt = 0;
struct object *Vector = 0;

struct object *Vector_new(struct closure *closure, struct Vector *self, int size)
{
  struct Vector *clone = (struct Vector *)send(self->_vt[-1], s_allocate, sizeof(struct Vector));
  clone->length   = size;
  clone->contents = (struct object *)calloc(size, sizeof(struct object *));
  return (struct object *)clone;
}

int Vector_length(struct closure *closure, struct Vector *self)
{
  return self->length;
}

struct Cons
{
  struct vtable *_vt[0];
  struct object *car;
  struct object *cdr;
};

struct vtable *Cons_vt = 0;
struct object *Cons = 0;

struct object *Cons_new(struct closure *closure, struct Cons *self, struct object *car, struct object *cdr)
{
  struct Cons *clone = (struct Cons *)send(self->_vt[-1], s_allocate, sizeof(struct Cons));
  clone->car = car;
  clone->cdr = cdr;
  return (struct object *)clone;
}

int Cons_length(struct closure *closure, struct Cons *self)
{
  return self->cdr
    ? 1 + (int)send(self->cdr, s_length)
    : 0;
}

void init2(void)
{
  s_new    = symbol_intern(0, 0, "new");
  s_length = symbol_intern(0, 0, "length");

  Number_vt = (struct vtable *)send(object_vt, s_delegated);
  String_vt = (struct vtable *)send(object_vt, s_delegated);
  Symbol_vt = (struct vtable *)send(object_vt, s_delegated);
  Vector_vt = (struct vtable *)send(object_vt, s_delegated);
  Cons_vt   = (struct vtable *)send(object_vt, s_delegated);

  send(Number_vt, s_addMethod, s_new, Number_new);
  send(String_vt, s_addMethod, s_new, String_new);
  send(Symbol_vt, s_addMethod, s_new, Symbol_new);
  send(Vector_vt, s_addMethod, s_new, Vector_new);
  send(Cons_vt,   s_addMethod, s_new,   Cons_new);

  send(Number_vt, s_addMethod, s_length, Number_length);
  send(String_vt, s_addMethod, s_length, String_length);
  send(Symbol_vt, s_addMethod, s_length, Symbol_length);
  send(Vector_vt, s_addMethod, s_length, Vector_length);
  send(Cons_vt,   s_addMethod, s_length,   Cons_length);
  Number = send(Number_vt, s_allocate, 0);
  String = send(String_vt, s_allocate, 0);
  Symbol = send(Symbol_vt, s_allocate, 0);
  Vector = send(Vector_vt, s_allocate, 0);
  Cons   = send(Cons_vt,   s_allocate, 0);
}

void doit(void)
{
  int i, j;

  struct object *a = send(String, s_new, 1);
  struct object *b = send(Symbol, s_new, a);
  struct object *c = send(Vector, s_new, 3);
  struct object *d = send(Cons,   s_new, 0, 0);

  for (i = 0, j = 0;  i < 1000000;  ++i)
    {
      j += (int)send(a, s_length) + (int)send(b, s_length) + (int)send(c, s_length) + (int)send(d, s_length);
      j += (int)send(a, s_length) + (int)send(b, s_length) + (int)send(c, s_length) + (int)send(d, s_length);
      j += (int)send(a, s_length) + (int)send(b, s_length) + (int)send(c, s_length) + (int)send(d, s_length);
      j += (int)send(a, s_length) + (int)send(b, s_length) + (int)send(c, s_length) + (int)send(d, s_length);
      j += (int)send(a, s_length) + (int)send(b, s_length) + (int)send(c, s_length) + (int)send(d, s_length);
      j += (int)send(a, s_length) + (int)send(b, s_length) + (int)send(c, s_length) + (int)send(d, s_length);
      j += (int)send(a, s_length) + (int)send(b, s_length) + (int)send(c, s_length) + (int)send(d, s_length);
      j += (int)send(a, s_length) + (int)send(b, s_length) + (int)send(c, s_length) + (int)send(d, s_length);
      j += (int)send(a, s_length) + (int)send(b, s_length) + (int)send(c, s_length) + (int)send(d, s_length);
      j += (int)send(a, s_length) + (int)send(b, s_length) + (int)send(c, s_length) + (int)send(d, s_length);
    }

  printf("total %d\n", j);
}

int main()
{
  init();
  init2();
  doit();
  return 0;
}
```

C. Equivalent benchmark using tagged `union` and `switch`

The example data types from the previous section implemented as a C tagged `union` with primitives that discriminate between them using a `switch` on the tag value stored in the `union`.

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

enum { Number, String, Symbol, Vector, Cons };

struct Number
{
};

struct String
{
  int   length;
  char *contents;
};

struct Symbol
{
  struct String *string;
};

struct Vector
{
  int            length;
  struct Object *contents;
};

struct Cons
{
  struct Object *car;
  struct Object *cdr;
};

typedef struct Object
{
  int tag;
  union {
    struct Number number;
    struct String string;
    struct Symbol symbol;
    struct Vector vector;
    struct Cons   cons;
  } payload;
} *oop;

static inline int length(struct Object *object)
{
  switch (object->tag)
    {
    case Number:
      fprintf(stderr, "Number has no length\n");
      exit(-1);

    case String:
      return object->payload.string.length;

    case Symbol:
      return object->payload.symbol.string->length;

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

    case Cons:
      return object->payload.cons.cdr
? 1 + length(object->payload.cons.cdr)
: 0;

    default:
      fprintf(stderr, "illegal tag %d\n", object->tag);
      exit(-1);
    }
}

int main()
{
  int i, j;
  struct Object *a= calloc(1, sizeof(struct Object));
  struct Object *b= calloc(1, sizeof(struct Object));
  struct Object *c= calloc(1, sizeof(struct Object));
  struct Object *d= calloc(1, sizeof(struct Object));

  a->tag= String;  a->payload.string.length= 1;
  b->tag= Symbol;  b->payload.symbol.string= (struct String *)a;
  c->tag= Vector;  c->payload.vector.length= 3;
  d->tag= Cons;    c->payload.cons.cdr= 0;

  for (i= 0, j= 0;  i < 1000000;  ++i)
    {
      j += length(a) + length(b) + length(c) + length(d);
      j += length(a) + length(b) + length(c) + length(d);
      j += length(a) + length(b) + length(c) + length(d);
      j += length(a) + length(b) + length(c) + length(d);
      j += length(a) + length(b) + length(c) + length(d);
      j += length(a) + length(b) + length(c) + length(d);
      j += length(a) + length(b) + length(c) + length(d);
      j += length(a) + length(b) + length(c) + length(d);
      j += length(a) + length(b) + length(c) + length(d);
      j += length(a) + length(b) + length(c) + length(d);
    }

  printf("total %d\n", j);

  return 0;
}
```