



# Quantum Object Dynamics

Ian Piumarta

**This material is based upon work supported in part by the National Science Foundation under Grant No. 0639876. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.**

VPRI Memo M-2009-002

# Quantum Object Dynamics

Ian Piumarta

21 September 2008

updated 27 September 2008

updated 15 January 2009

ian@vpri.org

## Introduction

This memo suggests that many interesting organisations and behaviours are founded on a single primitive operation: n-way associative lookup. An fast implementation of this primitive, in software or in hardware, could be the basis of very efficient implementations of a wide and diverse range of programming language semantics.

This suggestion could be argued top-down by choosing a range of interesting behaviours and organisations and showing how they decompose into a single primitive, or bottom-up by showing how a single primitive operation can be used alone or in recombination with itself to arrive incrementally at a number of familiar (and widely-used) behaviours. Since the latter seems more open-ended (and since I'm a bottom-up kind of person) I chose that approach.

## 1 Proposition

The static and dynamic characteristics of a wide range of object models can be described as trivial parameterisations of a simple multi-way content-addressable memory.

## 2 An abstract model of memory

For clarity we introduce an artificial distinction between application and primitive mechanisms. Application mechanisms are the fundamental semantic operations needed to implement some programming system (in Smalltalk we would identify dynamic binding as the critical semantic operation); they form the most essential part of the programming model exposed to users of that system (even if not always made directly available to those users). The primitive mechanism is the raw material used by the system implementor as a platform on which to build the application semantics (in most Smalltalk implementations we would have to admit that the primitive mechanisms are memory allocation and base+offset addressing of that memory).

Our primitive mechanism provides a memory that is a map  $m$  associating one or more keys  $k_i$  with a value  $v$ .

$$m : \mathbb{K}^* \rightarrow \mathbb{V}$$
$$m[k_1, \dots, k_n] = v$$

This memory supports two primitive operations, associative read and write.

$$m[k_1, \dots, k_n] \quad \text{denotes the value in } m \text{ associated with the keys } k_i$$
$$m[k_1, \dots, k_n] \triangleleft v \quad \text{denotes in-place modification of } m; \text{ subsequently } m[k_1, \dots, k_n] = v$$

(The notation  $m[k]$  is used instead of  $m(k)$  to remind us that  $m$  is not a function in the mathematical sense.) The state of  $m$  is therefore relative to a particular time, the passage of which will be implied but not stated in this discussion.<sup>1</sup>

---

<sup>1</sup>Object models in which time, versioning, causality, etc., are significant are far better described if their time component is just another key, in the range of user-accessible values, rather than a fundamental property of the abstract model.

The domain  $\mathbb{K}$  of keys and range  $\mathbb{V}$  of values in  $m$  are the same. A distinguished value  $\epsilon$  (the “undefined” value) is initially associated with every possible combination of keys in  $m$ . If  $\epsilon$  is used as a key, the associated value is also  $\epsilon$  (regardless of the other keys).

$$m[k_1, \dots, k_n] = \epsilon \quad \text{for any } k_i = \epsilon$$

A simple model might signal a runtime error whenever an  $\epsilon$  value is read.

Application mechanisms are presented as read and write operations on memory via the functions  $r$  and  $w$ , respectively.

$$\begin{aligned} r : \mathbb{K}^* &\rightarrow \mathbb{V} && \text{reads a value } v \text{ in } m \text{ associated with } k_i \\ w : \mathbb{K}^* \times \mathbb{V} &\rightarrow \epsilon && \text{writes to } m \text{ such that subsequently } r(k) = v \end{aligned}$$

For a given programming system we are therefore interested in defining its ‘characteristic’ functions  $r$  and  $w$  of  $k$  in terms of the two primitive operations  $\sqcup$  and  $\sqtriangleleft$  on  $m$ . To see how this works, consider the simplest possible object model: a flat address space...

### 3 Physical memory

Reading a memory address yields a value; writing a memory address updates its value. The functions  $r$  and  $w$  are trivially defined as the two fundamental operations on  $m$ .<sup>2</sup>

$$\begin{aligned} r(k) &= m[k] \\ w(k, v) &= m[k] \sqtriangleleft v \end{aligned}$$

### 4 Various kinds of dictionary

Two addresses  $k_1$  and  $k_2$  are associated with each value.

$$\begin{aligned} r(k_1, k_2) &= m[k_1, k_2] \\ w(k_1, k_2, v) &= m[k_1, k_2] \sqtriangleleft v \end{aligned}$$

This describes arrays and traditional structures. (The latter are arrays whose indices are encodings of the structure field names written by the programmer.) Furthermore, with suitable initialisation of  $k_2$  for each  $k_1$  “created”,  $r$  and  $w$  implement virtual memory in which  $k_1$  is a segment identifier and  $k_2$  is an address within  $k_1$ . At sufficiently small granularity,  $k_1$  is an identifier for a strongly-encapsulated “object” and  $k_2$  a field designator within that object. (In other words,  $k_1$  identifies a “dictionary” object and  $k_2$  a “slot” within that dictionary.)

Hardware implementations of this kind of memory (with bounded keys) have treated  $k_1$  as a base address,  $k_2$  as an offset, and  $\epsilon$  as an access violation. A more dynamic representation (such as an object or hash table) would be appropriate if the keys  $k_1$  and/or  $k_2$  are potentially unbounded (generated arbitrarily by the running program, for example).

### 5 Recursively-defined semantics

Instead of signalling an error (or being converted to some default application value) we will let reading  $\epsilon$  from  $m$  cause the read operation to be restarted with the original keys transformed by a set of functions  $\beta_i$ .

$$r(k_1, k_2) = \begin{cases} m[k_1, k_2] & \text{for } m[k_1, k_2] \neq \epsilon \\ r(\beta_1(k_1), \beta_2(k_2)) & \text{for } m[k_1, k_2] = \epsilon \end{cases}$$

---

<sup>2</sup>In a  $w$ -bit computer (with no memory paging or segmentation) we might have  $\mathbb{K} = \{n \in \mathbb{N}_0 \mid 0 \leq n < 2^w\}$ . This vastly underutilises the power of the primitive mechanism, but would work.

In other worlds, the  $\beta$  functions repeatedly transform the combination of keys  $k_i$  to find an associated non- $\epsilon$  value in  $m$ .

Let  $\sigma$  denote a particular well-known key, distinct from any other application key. One useful set of  $\beta$ -transformations is

$$\begin{aligned}\beta_1(k) &= m[k, \sigma] \\ \beta_2(k) &= k\end{aligned}$$

that, when substituted back into  $r$ , yield

$$r(k_1, k_2) = \begin{cases} m[k_1, k_2] & \text{for } m[k_1, k_2] \neq \epsilon \\ r(m[k_1, \sigma], k_2) & \text{for } m[k_1, k_2] = \epsilon \end{cases}$$

which *delegates* the lookup of the “slot”  $k_2$  within the “object”  $k_1$  to the object stored as the value of the  $\sigma$  slot in  $k_1$ , whenever  $k_1$  has no  $k_2$  slot of its own. In the case that  $k_2$  is being used as a message name then the above describes the dynamic binding part of the method lookup operation in delegation-based message passing.

Expressing this delegation as a pair of  $\beta$  transformations on keys in an associative memory emphasizes a fundamental symmetry of the delegation mechanism that is ignored by most object-oriented programming languages:

- if  $\beta_1(k) = m[k, \sigma]$  and  $\beta_2(k) = k$  then several different objects  $k_1$  delegate between themselves the search for a non- $\epsilon$  value associated with a particular slot name  $k_2$ ;
- if  $\beta_1(k) = k$  and  $\beta_2(k) = m[k, \sigma]$  then several different slot names  $k_2$  delegate between themselves the search for a non- $\epsilon$  value within a particular object  $k_1$ .

Combining delegation along both of these “axes” within a two-dimensional delegation space  $k_1 \times k_2$  provides the basis for simple (but powerful) sharing and protection mechanisms.

The above  $\beta$  functions are “post-lookup” transformations. It is also useful to consider “pre-lookup” transformations. Consider a set of functions  $\alpha_i$  that are applied to  $k_i$  producing a transformed set of keys to which  $r$  is then applied.

$$\begin{aligned}r(k_1, k_2) &= r'(\alpha_1(k_1), \alpha_2(k_2)) \\ r'(k_1, k_2) &= \begin{cases} m[k_1, k_2] & \text{for } m[k_1, k_2] \neq \epsilon \\ r'(\beta_1(k_1), \beta_2(k_2)) & \text{for } m[k_1, k_2] = \epsilon \end{cases}\end{aligned}$$

Let  $\tau$  denote a particular well-known key, distinct from any other application key. One useful set of  $\alpha$ -transformations is

$$\begin{aligned}\alpha_1(k) &= m[k, \tau] \\ \alpha_2(k) &= k\end{aligned}$$

that, when substituted back into  $r$  (keeping the  $\beta$ -transformations of the delegation example), yield

$$\begin{aligned}r(k_1, k_2) &= r'(m[k_1, \tau], k_2) \\ r'(k_1, k_2) &= \begin{cases} m[k_1, k_2] & \text{for } m[k_1, k_2] \neq \epsilon \\ r'(m[k_1, \sigma], k_2) & \text{for } m[k_1, k_2] = \epsilon \end{cases}\end{aligned}$$

which uses some “property”  $\tau$  of an object  $k_1$  as the starting point for the previous example’s lookup (following a “chain” of  $\sigma$  slots). Put another way, if  $k_2$  is interpreted as a message name then  $\tau$  is the “type” of an object (grouping related objects into a family) and  $\sigma$  the “supertype” of a type. In other words

$$\begin{aligned}n &= 2 \\ \alpha_1(k) &= m[k, \tau] \\ \beta_1(k) &= m[k, \sigma]\end{aligned}$$

is the dynamic binding mechanism for a class-based object system with inheritance.<sup>3</sup>

## 6 Keys are meta-taxonomic dimensions

Each particular well-known key, along with its recursive  $\beta$ - and  $\alpha$ -transformations, can generate a taxonomy within which objects in the memory can be organised. In the above examples, applied to a Smalltalk-like system,  $\tau$  is an object's class pointer and  $\sigma$  is a (meta)class' superclass pointer (both of which are hierarchical taxonomies). Each is associated with a different concrete key, but both exist in the same dimension (are used in the same position  $k_i$ , where  $i = 2$  in this case).

Each additional key position (gained by increasing  $n$  by 1, for example) creates a new “dimension” or “taxonomic space” in which any number of new taxonomies can be created. These new taxonomies will all be orthogonal to (and completely independent from) those in other key positions (even if they share the same concrete keys).

Continuing with the delegation example, increasing  $n$  to 3 (adding the key  $k_3$ )

$$r(k_1, k_2, k_3) = \begin{cases} m[k_1, k_2, k_3] & \text{for } m[k_1, k_2, k_3] \neq \epsilon \\ r(m[k_1, \sigma], k_2, k_3) & \text{for } m[k_1, k_2, k_3] = \epsilon \end{cases}$$

gives us multiple (disjoint) perspectives on objects, each associated with a particular concrete  $k_3$ , with delegation occurring between objects only within a single perspective. In effect,  $k_3$  is a ‘namespace’ constraining both the content of, and the extent of the taxonomies defined by concrete keys and their  $\beta$  functions between, objects ‘residing’ within it.

If we have a namespace  $\omega$  in which global relationships are expressed and let

$$\beta_3(k) = m[k, \sigma, \omega]$$

then perspectives (the  $k_3$  keys) on a given object will delegate to each other (via their  $\sigma$  slot).

The occurrence of  $\epsilon$  in  $m$  can be used to terminate delegation (or other recursive relationships) in multiple dimensions. Introducing distinct versions of  $r$  (one  $r_i$  for each dimension  $i$  in which delegation occurs) lets us choose the precedence of axes in the  $n$ -dimensional delegation space. For example,

$$r(k_1, k_2, k_3) = \begin{cases} r_1(k_1, k_2, k_3) & \text{for } r_1(k_1, k_2, k_3) \neq \epsilon \\ r_1(k_1, k_2, m[k_3, \sigma]) & \text{otherwise} \end{cases}$$

$$r_1(k_1, k_2, k_3) = \begin{cases} m[k_1, k_2, k_3] & \text{for } m[k_1, k_2, k_3] \neq \epsilon \\ r_1(m[k_1, \sigma], k_2, k_3) & \text{otherwise} \end{cases}$$

delegates first between objects  $k_1$  within a single perspective  $k_3$  and then between perspectives  $k_3$  on the original object, whereas

$$r(k_1, k_2, k_3) = \begin{cases} r_3(k_1, k_2, k_3) & \text{for } r_3(k_1, k_2, k_3) \neq \epsilon \\ r_3(m[k_1, \sigma], k_2, k_3) & \text{otherwise} \end{cases}$$

$$r_3(k_1, k_2, k_3) = \begin{cases} m[k_1, k_2, k_3] & \text{for } m[k_1, k_2, k_3] \neq \epsilon \\ r_3(k_1, k_2, m[k_3, \sigma]) & \text{otherwise} \end{cases}$$

delegates first between perspectives  $k_3$  on a single object  $k_1$  and then between distinct objects  $k_1$  in the original perspective  $k_3$ .

One final example (among many): if we let  $v$  range over methods of arity  $n$  within a memory indexed by  $k_1, \dots, k_n$ , then the above model (with appropriate  $\alpha$ - and  $\beta$ -transformations) can easily describe binding mechanisms for multimethod (generic function) dispatch.

---

<sup>3</sup>Of course, not all the complexity is contained within the three lines that characterise the mechanism. For example, they only tell you some of what you need to know to be able to create the initial class hierarchy, install new methods in classes, or implement a `ClassBuilder` in Smalltalk.

## 7 The function $w$ and its transformations

These are constructed in exactly the same manner as for the function  $r$ , with the same possibilities for pre- and post-transformations and for recursive recombination, in the obvious manner.

The simplest useful definition of the application write function

$$w(k_i, \dots, k_n, v) = m[k_i, \dots, k_n] \triangleleft v$$

introduces new keys into  $m$  directly with no attempt to reason about “where” the new value  $v$  should be “placed” within any taxonomies defined by  $r$ . In the same manner as was done for  $r$ , pre-transformations  $\gamma_i$  and post-transformations  $\delta_i$  can be introduced.

$$w(k_1, \dots, k_n) = w'(\gamma_1(k_1), \dots, \gamma_n(k_n))$$
$$w'(k_1, \dots, k_n) = \begin{cases} w'(\delta_1(k_1), \dots, \delta_n(k_n)) & \text{for some condition involving } \epsilon, r, \alpha_i, \beta_i, \gamma_i, \delta_i \\ m[k_1, \dots, k_n] \triangleleft v & \text{otherwise} \end{cases}$$

It is worthwhile to note that this “simplest useful” definition of  $w$  is almost always the most appropriate. (For the inheritance and delegation mechanisms described above it is precisely what is wanted.) More exotic constructions for  $w$  would be identical in nature to those already examined for the function  $r$ .

## 8 Unification

The primitive read and write operations on  $m$  can be unified into a single operation. To write a value  $v$ , a statement

$$m[k_1, \dots, k_n, v]$$

is made about its presence within the memory. (If  $v = \epsilon$  the value is “deleted”.) Unifying a single variable  $v$  within a similar statement

$$m[k_1, \dots, k_n, v]$$

retrieves a value. It is trivial to rephrase this entire memo using the above formulation.

This simplification suggests a very powerful extension that would allow the ‘unified’ variable(s) to appear in any key position, not just the last—true content-addressability. The primitive mechanism is now directly applicable to the semantics of local operations of relational languages.<sup>4</sup> (Generalised support for publish-subscribe would then require ‘just’ the addition of a global notification mechanism. One possibility might be ‘future unification’ where a process blocks until a non- $\epsilon$  value becomes available for each unified variable in a statement.)

Such extensions are not without practical and philosophical costs (far beyond the already considerable implementation challenges presented by the basic primitive mechanism).

## 9 Practical considerations

Some of the application-level models of organisation and dynamic behaviour described in this memo are trivial to implement on (or are intrinsic to) current computer hardware. All of them are trivial to implement given the primitive  $\square$  and  $\square \triangleleft$  operators. Furthermore, if these implementations are efficient then the resulting programming system will be efficient, with complexity increasing commensurately (in the absolute worst case exponentially) with  $n$ .

Software implementations for all of the models/behaviours presented for are common for  $n = 2$ , and can be made very efficient (through various caching techniques) for  $\alpha_i$  that map many objects onto a much smaller

---

<sup>4</sup>Looking at it from the other direction: an efficient relational language is sufficient to implement all of the mechanisms described in this memo.

set of object families. Hash tables work well for ‘singleton’ associations where  $n = 2$  and  $alpha(k) = k$ , but already present problems of garbage collection: values should be deleted from  $m$  when either  $k_1$  or  $k_2$  becomes unreachable, but it is usual to consider only  $k_1$ . The problem becomes increasingly difficult as generality is preserved while  $n$  grows beyond 2, where unreachability of any given key  $k$  must imply deletion of all values for which some  $k_i = k$  (for any  $i : 0 \leq i < n$ ). It seems clear that some cooperation between the primitive mechanism and end-user storage management collector is required, since the latter almost certainly places implicit constraints on the combinations of values stored in the memory that would simplify (or even make possible) primitive storage management.

In some models the storage management should participate in simplifying end-user structures when keys vanish. For example, given three keys  $k_a, k_b$  and  $k_c$  for which a model defines a  $\beta(k)$  as

$$\begin{aligned} \beta_1(k) = m[k, \sigma] & \quad \text{such that} \\ m[k_a, \sigma] = k_b & \quad \text{and} \\ m[k_b, \sigma] = k_c & \end{aligned}$$

then the unreachability of  $k_b$  (which occurs between  $k_a$  and  $k_c$  in a transitive relationship) should cause all values in  $m$  associated with  $k_b$  to be re-associated with  $k_a$ , and the relationship between the keys simplified to

$$m[k_a, \sigma] = k_c$$

Hardware support for large memories with unconstrained (or a relatively large limit on)  $n$  would enable efficient implementations of a wide variety of interesting object models, both commonly used and many yet to be imagined. I am not a hardware expert, but I believe that current virtual memory hardware is not far from being useful (if it could be scaled) for this purpose.

## 10 Conclusion

This memo is an attempt to stimulate thinking about how a very simple pair of primitive operations (that should be efficiently realisable in sufficiently parallel hardware) can scale to (and adequately implement with trivial additional work) the complex structures and behaviours we struggle to implement in object-oriented, functional and relational systems.

I hope it also managed to demonstrate that many apparently very different and interesting organisations and behaviours are in fact closely related as slight variations within a general parameterisable n-way associative memory. (A suitably bored CS student could probably, in short order, characterise the fundamental operations in dozens of seemingly disparate languages in terms of simple  $\beta$  and  $\alpha$  functions.)

I have no idea what hardware support can be easily made for the primitive operators within familiar architectural components (such as virtual memory). I do know that an efficient software implementation, capable of scaling to billions of entries, would make a great thesis. The big challenges are not necessarily to be found in the primitive operators, but rather in the associated management—garbage collection in particular.

I believe that a hardware company wondering how software will ever be able to make good use of their trillion (and growing) transistors per die should look seriously at massively parallel support for n-way associative memory. While it may be only marginally useful as a raw programming model, a hardware scheme providing the two operators described in this memo would clearly make a huge difference in the implementation of many popular dynamic languages.