An association-based model of dynamic behaviour*

Ian Piumarta
Viewpoints Research Institute, Glendale, CA, USA
ian@vpri.org

ABSTRACT

Dynamic programming languages seem to spend much of their time looking up behaviour associatively. Data structures in these languages are also easily expressible as associations. We propose that many, and maybe even all, interesting organisations of information and behaviour might be built from a single primitive operation: n-way associative lookup. A fast implementation of this primitive, possibly in hardware, could be the basis of efficient and compact implementations of a diverse range of programming language semantics and data structures.

1. INTRODUCTION

Languages with dynamic dispatch [9], first-class environments, and similar late-binding mechanisms, use associative lookup as a central component of the mechanisms and semantics they provide. For example, message sending (or calling a virtual function) uses an association from types and message (or function) names to function implementations. Multiple dispatch [2] typically associates a sequence of several (potentially many) type names with a function or method implementation. Even simple objects with named fields, or indexable arrays, are associations between an object identifier and a field name or numeric index that need not specify further how the storage is implemented.

This leads to the question of whether many (maybe even all) useful organisations of information and behaviour in a dynamic language might be constructed from a single primitive operation: n-way associative lookup.

We could explore this question top-down by choosing a range of interesting behaviours and organisations and showing how they can be composed from a single primitive, or bottom-up by showing how a single primitive operation can be used alone or in composition with itself to arrive incrementally at a number of familiar and widely-used behaviours and organisations. Since the latter seems more open-ended, that is the approach we take here.

*This material is based upon work supported in part by the National Science Foundation under Grant No. 0639876. Opinions, findings, and conclusions or recommendations expressed in this material are those of the author and almost certainly do not reflect those of the NSF—or of anyone else, for that matter.

FREECO'11, July 26, 2011, Lancaster, UK

2. AN ABSTRACT MODEL OF MEMORY

We will distinguish between application and primitive mechanisms.

Application mechanisms are the fundamental semantic operations needed to implement some programming system. In Smalltalk [5], for example, we would identify dynamic binding as the critical semantic operation. These mechanisms form the essential part of the programming model exposed to users of the system, even if they are not always made directly available to users.

Primitive mechanisms are the raw material used by the language 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 (hidden within primitives `new` and `new:`) and base+offset addressing of that memory (hidden within the various primitives `at:` and `at:put:`).

2.1 Primitive mechanism

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

$$m : \mathbb{K}^* \rightarrow \mathbb{V}$$

$$m[k_1, ..., k_n] = v$$

This memory supports two primitive operators, associative read and associative write, which we will be written as '[]' and '[]◁' respectively:

$m[k_1, ..., k_n]$	value in m associated with keys k_i
$m[k_1, ..., k_n] \lhd v$	update m; subsequently $m[k_i] = v$

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

The domain $\mathbb{K}$ of keys and range $\mathbb{V}$ of values in m are the same. A distinguished value ϵ (the "undefined" value) is initially associated with every possible combination of keys

[1]Object models in which time, versioning, causality, etc., are significant are probably far better modelled by considering the time component as another key (a first-class user-accessible value) rather than an intrinsic property of the underlying model.

in m. If ϵ is used as a key, the associated value is also ϵ (regardless of the other keys).

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

A simple application model might choose to let an ϵ value propagate through subsequent operations, or to raise an exception immediately when an ϵ is read, etc.

2.2 Application mechanism

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

$$\begin{aligned} r &: \mathbb{K}^* \to \mathbb{V} \\ w &: \mathbb{K}^* \times \mathbb{V} \to \epsilon \end{aligned}$$

$r(k_1, ..., k_n)$	read value associated with keys k_i
$w(k_1, ..., k_n, v)$	write value associated with keys k_i

For a given object model we would like to define its 'characteristic' functions r and w of k in terms of the primitive operations $[]$ and $[]\triangleleft$. To illustrate this we will 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.

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

In a w-bit computer (with no paging or segmentation) we might have $\mathbb{K} = \{n \in \mathbb{N}_0 \mid 0 \leq n < 2^w\}$, which is just the memory addresses representable by a w-bit word. This flat address space model completely ignores the useful properties of the primitive mechanism, but would work.

4. DICTIONARIES

Two addresses k_1 and k_2 are associated with each value.

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

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 ϵ 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 ϵ from m cause the read operation to be restarted with the original keys transformed by a set of functions β_i.

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

In other words, the β functions repeatedly transform the combination of keys k_i to find an associated non-ϵ value in m.

Let σ denote a particular well-known key, distinct from any other application key. One useful set of β-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* [7] the lookup of the "slot" k_2 within the "object" k_1 to the object stored as the value of the σ 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 β-transformations on keys in an associative memory emphasises 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-ϵ 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-ϵ 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 several kinds of useful encapsulation and sharing mechanisms, capable of addressing problems such as behavioural inheritance [8] in a dynamic and open system.[2]

The above β functions are "post-lookup" transformations. It is also useful to consider "pre-lookup" transformations. Consider a set of functions α_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 τ denote a particular well-known key, distinct from any other application key. One useful set of α-transformations is

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

[2]For example, if slots contain named methods then 'selector inheritance' allows multiple incompatible versions of methods to coexist without interference. In that case, delegation should occur between types before names, such that a particular specialisation of behaviour would have the chance to benefit from 'normal' inheritance between its own overridden definitions of methods (along the type axis) before delegating along the name axis (after falling off the end of the supertype chain) to the 'base' set of methods that implement the original behaviour. Delegation along multiple axes is not meant to imply any kind of concurrent or nondeterministic mechanism.

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

$$r(k_1, k_2) = r'(m[k_1, \tau], k_2)$$

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

which uses some "property" τ of an object k_1 as the starting point for the previous example's lookup (following a "chain" of σ slots). Put another way, if k_2 is interpreted as a message name then τ is the "type" of an object (grouping related objects into a family) and σ 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.

Of course, not all the complexity of a practical system is contained within the three lines that characterise the mechanism. For example, in Smalltalk these three lines say nothing about creating the initial class hierarchy, installing new methods in classes, or implementing a `ClassBuilder` object.

6. KEYS ARE META-TAXONOMIC DIMENSIONS

Each particular well-known key, along with its recursive β- and α-transformations, can generate a taxonomy within which objects can be organised. In the above examples, applied to a Smalltalk-like system, τ is an object's class pointer and σ is a superclass pointer in a (meta)class (both of which are hierarchical taxonomies of object types). 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 β functions between, objects 'residing' within it.

If we have a namespace ω 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 σ slot).

The occurrence of ϵ 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 $k_1, ...k_n$, then the above model (with appropriate α- and β-transformations) can easily describe binding mechanisms for multi-method (generic function) dispatch.

7. 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 w, the application write function,

$$w(k_i, ..., k_n, v) = m[k_i, ..., k_n] \lhd v$$

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 γ_i and post-transformations δ_i can be introduced.

$$w(k_1, ..., k_n) = w'(\gamma_1(k_1), ..., \gamma_n(k_n))$$

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

It is worthwhile to note that this "simplest useful" definition of w is often 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

$$m[k_1, ..., k_n, v]$$

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

$$v = m[k_1, ..., k_n, ?]$$

retrieves a value. It is trivial to rephrase this entire paper 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. The primitive mechanism is now directly applicable to the semantics of local operations of relational languages.[3] (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-ϵ 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 paper are trivial to implement on (or are intrinsic to) current computer hardware. All of them are trivial to implement given the primitive $[\,]$ and $[\,]\triangleleft$ 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 α_i that map many objects onto a much smaller 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

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

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. Current virtual memory hardware might not be far from useful, if it could be scaled, for this purpose.

Nothing has been said here about several facilities that were tacitly assumed throughout the paper:

- access to non-key data: bytes, integers, floats, etc.;
- arithmetic and logical operations on keys and non-key data;
- choice mechanisms, to select between alternatives in the piecewise definitions or 'otherwise' clauses;
- many other kinds of program sequencing and data manipulation that would be present in a general programming system.

One question is whether the mechanisms described in this paper can be efficiently supported by a relational database engine [3]. Simple experiments with an in-memory Sqlite [10] database[4] suggests not, with dynamic binding operations taking several orders of magnitude longer than would be possible with a dedicated implementation typical of a language runtime support library. In the absence of hardware, the approach presented in this paper could still be valuable as a means to present and compare dynamic behaviours, or as a concise notation for dynamic mechanisms to be compiled and run on stock architectures as part of the language runtime support library.

10. DISCUSSION

A dynamic language implementation built indirectly, using the association primitive described above and without hardware acceleration, is unlikely to be more efficient than a direct implementation. The direct implementation would use algorithms either equivalent to the association primitive or (more likely) specialised and optimised for the particular language in ways that are inappropriate for a generic primitive, and therefore more efficient.

Independent of any possibility for hardware support, the association primitive is useful to understand the semantics and complexity of a particular application mechanism and to compare and contrast different mechanisms. One kind of question that might be more easily answered is whether two dynamic mechanisms are fundamentally different, or whether they are the result of breaking the symmetries of a generic mechanism in two different ways. (The symmetry between type-centric and selector-centric inheritance, described in Section 5, is one example.) Understanding and comparing different mechanisms as specialisations of a general common mechanism might also suggest possibilities for new combinations of mechanisms.

Section 6 already suggests a general mechanism that can be parameterised by number of axes (or dimensions) involved in lookups. As pointed out in [6], one way to choose the meaning of successive axes results in a progression from object-oriented to subject-oriented to context-oriented programming. Similarly, Worlds [13] is a simple addition of an axis to an existing association/aggregation mechanism. We could continue to invent new and exciting mechanisms beyond context-orientation and Worlds, *ad absurdum*, or recognise they are all just points within a dimensionally unbounded space of related, self-similar mechanisms.

[3]Looking at it from the other direction: an efficient relational language is sufficient to implement all of the mechanisms described in this memo.

[4]Sqlite has the fastest in-memory implementation of the freely-available relational databases.

This suggests using the association primitive as part of a language definition, to be translated automatically into an efficient implementation. The dynamic mechanisms with which we are familiar (as described above, as well as those that have not been invented yet) could, and probably should, be nothing more remarkable than the consequences of particular arrangements of implications made from properties of objects described by the programmer as part of the specification of the environment in which their application program will be written and executed.[5]

11. RELATED WORK

Context-oriented programming [6] addresses similar issues, but provides solutions at a much higher level of abstraction by extending high-level languages (Lisp and Java) within themselves to add another axis to the binding process.

Predicate dispatch [4] unifies many mechanisms for choosing dynamically between methods within a generic function, but is qualitatively different to the present approach in its heavy reliance on compile-time static type analysis.

Maybe the most closely-related type-based work is $\lambda^{\{\}}$ [1] which also seeks to unify the overloading of methods to form generic functions, but does so dynamically and tries to use the smallest number of operators to accomplish the task.

In contrast to the above, the association primitive is simpler since it considers type as an optional runtime property derived from a value, not as a formal property of an abstract value at compile time.

12. CONCLUSION

This paper 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.

Hopefully it also manages 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.

We may never see hardware support for the primitive operators described here, but an efficient software implementation (capable of scaling to billions of entries) would make a great doctoral thesis. The big challenges are not necessarily to be found in the primitive operators, but rather in the associated management—garbage collection, in particular.

13. REFERENCES

[1] G. Castagna (1997) *Unifying overloading and λ-abstractions: $\lambda^{\{\}}$*, Theoretical Computer Science, Vol. 176, No. 1–2, pp. 337–345

[2] C. Chambers (1992) *Object-Oriented Multi-Methods in Cecil*, Proc. European Conference on Object-Oriented Computing (ECOOP'92), pp. 33–56

[3] E. Codd (1970) *A relational model of data for large shared data banks*, Communications of the ACM, Vol. 13, No. 6, pp. 377–387

[4] M. Ernst, C. Kaplan and C. Chambers (1998) *Predicate dispatching: A unified theory of dispatch*, Proc. 12th European Conference on Object-Oriented Programming (ECOOP'98), pp. 186–211

[5] A. Goldberg and D. Robson (1983) *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, ISBN 0–201–11371–6

[6] R. Hirschfeld, P. Costanza and O. Nierstrasz (2008) *Context-oriented Programming*, Journal of Object Technology (JOT), Vol. 7, No. 3, pp. 125–151

[7] H. Lieberman (1986) *Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems*, Proc. First ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA), Portland, OR

[8] B. Liskov and J. Wing (1994) *A behavioral notion of subtyping*, ACM Transactions on Programming Languages and Systems (TOPLAS), Vol. 16, No. 6, pp. 1811–1841

[9] S. Milton and H. Schmidt (1994) *Dynamic Dispatch in Object-Oriented Languages*, Technical Report TR–CS–94–02, Commonwealth Scientific and Industrial Research Organisation (CSIRO), Division of Information Technology

[10] `http://www.sqlite.org`

[11] A. Stepanov and M. Lee (1994) *The Standard Template Library*, Technical Report X3J16/94–0095, WG21/N0482, ISO Programming Language C++ Project

[12] B. Stroustrup (1997) *The C++ Programming Language*, Addison Wesley, ISBN 0–201–88954–4

[13] A. Warth, Y. Ohshima, T. Kaehler and A. Kay (2010) *Worlds: Controlling the Scope of Side Effects*, Technical Report TR–2010–001, Viewpoints Research Institute

Acknowledgements

The author is greatly indebted to the three anonymous reviewers who provided much useful feeback and who worked valiantly in hope of turning this paper into something of academic value. Responsibility for failure to achieve that goal rests entirely with the author.

[5] This "meta programming" should be no more intimidating than the current "meta" practices of defining a template library [11], or redefining operators `new` and `delete`, to extend the environment in which C++ [12] programs are written, for example.