On Owners-as-Accessors

James Noble and Alex Potanin

Victoria University of Wellington, Wellington, New Zealand
{kjx,alex}@ecs.vuw.ac.nz

Abstract. Prescriptive ownership systems generally impose one of two disciplines on programs: either owners-as-dominators, or owners-as-modifiers. In this paper we discuss "*owners-as-accessors*" — a discipline that is stricter than owners-as-modifiers but more lenient than owners-as-dominators. We provide a concise informal definition of owners-as-accessors, discuss some existing systems that employ this discipline, and revisit an earlier study on performance. Finally we hypothesize how owners-as-accessors could potentially unify William Cook's two forms of data abstraction: abstract data types and objects.

Keywords: ownership · dominators · modifiers · accessors · readers · object-oriented programming

1 Introduction

Ownership Types were originally devised to implement flexible alias protection, that is, to guarantee the encapsulation of abstractions in object-oriented programs [13, 36]. Clarke's original ownership types adopted the "owners-as-dominators" discipline, restricting a program's heap topology so that all paths to an object from the root of the system passed through that object's *owner* — the "no incoming pointers" rule [40, 15]. A long line of research followed, developing new twists on precisely which heap topologies would be permitted, which pointers counted as incoming, and which should be forbidden [2, 14, 6, 37] — Clarke et al. [12] provide a magisterial survey of this and much subsequent work, much more than we can do justice to here. These relatively strict, mostly explicit, generally prescriptive, primarily static ownership systems are now being incorporated into practical languages, typically to manage concurrency [5, 26, 22] or memory allocation [41].

Addressing program verification rather than memory management, Müller [33] developed the "owners-as-modifiers" discipline as a significant relaxation of owners-as-dominators. Owners-as-modifiers does not restrict the heap topology *per se*, rather it distinguishes between reading and writing (modifying): any heap pointer may be traversed for reading, but only references from owners for writing [30, 18, 4]. Owners-as-modifiers permits many programming patterns that owners-as-dominators does not, however it no longer protects abstractions against encapsulation breaches, because any data that is reachable can always be read.

As Clarke et al. describe, development of static ownership types systems went on to newer [9, 3, 37] and weirder topologies [8, 11, 4], and substantial progress was made by many researchers in ownership inference and other static and dynamic analyses (again, see Clarke et al. [12] for an extensive account).

2 Owners-as-Accessors

As well as this "mainline" of work — strict owners-as-dominators for memory use; relaxed topologies for encapsulation or effects; more flexibility for heap analysis; and owners-as-modifiers for verification — there have been a few approaches that have taken a different tack to ownership, focusing on use rather than references.

An early example is Skalka and Smith's "Use-Based" ownership [43] that does not restrict heap topology but (somewhat akin to owners-as-modifiers) restricts the methods that incoming references can call on an object. Unlike owners-as-modifiers, this restriction is not based on a reading vs. writing distinction but rather arbitrary "whitelists" of methods invocable on objects from particular protection domains.

An even earlier example (although it was not obvious at the time) was a dynamic version of ownership also inspired by Flexible Alias Protection [35, 23]. In these kinds of systems, the ownership checks are dynamic: in Dynamic Ownership in particular, ownership would be checked on the receipt of every message by an object (i.e. at every method call) — and incoming method calls, rather than incoming references (or incoming modifications) raised a dynamic error. Wernli et al. have recently both implemented and formalised an extended version of Dynamic Ownership in Smalltalk [45].

Independently, Gruber and Boyer [24] developed a novel ownership discipline for actor isolation. Unlike other ownership based approaches that fully encapsulate actors, preventing both incoming and outgoing references [25, 44], Gruber and Boyer permit references to objects owned by other actors, but check dynamically on each access – to quote: "*we can allow illegal references to exist as long as we forbid their use*". An actor can pass a reference to an object from its own internal state to another actor, sometime later the other actor can pass the reference back again, when it can be be freely accessed by the original actor.

Equally independently, Dimoulas et al. [20] adapted an ownership system for blame [19] to handle object-capability security. Their capability system enforces that "*a component can directly [use a capability] only if it owns the capability or a guard for the capability*". In this system, a component owning a capability (i.e. a reference to another object) is a typical static ownership check, and the guard is an inserted dynamic ownership check.

The key point of this paper is that all of these various systems — in their quite different ways — implement the same ownership discipline which we have called *owners-as-accessors* [39]. Owners-as-accessors does not restrict the heap topology (neither does owners-as-modifiers), but nor does owners-as-accessors distinguish between reading and writing (neither does owners-as-dominators). Rather, owners-as-accessors requires all accesses (reading and writing) to be made via an object's owner.

Lightly editing Clark et al.'s pithy characterisation of Dietl's formulation of owners-as-modifiers [12, 18]: we can now define owners-as-accessors as ensuring that:

> "*no **access** to an object can occur unless the object's owner appears as the target of that method call on the stack*"

3 Programming with Owners-as-Accessors

We have conducted an empirical study of programming with owners-as-accessors [39] — although without providing either the clear definition we've introduced above, or any description of extant systems following this discipline. In this section we briefly revisit that study, and extend the data analysis.

Our study measured the performance of iterators, probably the most famous bugbear of owners-as-dominators, and certainly an inspiration for many extensions of permitted topologies [34, 7, 37]. Traditional external iterators [21] are not permitted under owners-as-dominators because they rely on incoming references (see fig. 1: the iterator holds an incoming pointer to a link inside the list, and can follow a link's pointers to the next or previous link). Where the underlying data structure provides $O(1)$ random access, this is not a problem, but where the underlying data structure only provides $O(N)$ access, the cost of iteration can rise from $O(N)$ to $O(N^2)$. By forcing such changes in designs, ownership can indirectly impose overheads on programs even if e.g. static checks (as typically used in owners-as-dominators) don't impose any overhead directly. (Alternatively, programmers could eliminate the encapsulation provided by ownership, by granting the same access to a collection's implementation as to the collection itself: in our study we aimed to maximise the encapsulation of collection implementations).

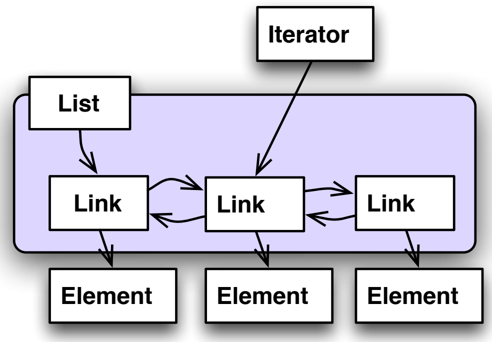

Fig. 1. Linked List with Iterator from [39]

The core of our study compared external iterators with iteration implemented via the Memento pattern [21] as used in Dylan [42]. In this design, iteration over a collection begins by creating an opaque "iteration state object" (the memento) and proceeds by passing that memento back into the collection at each step (see fig. 2: when the next element is requested from the memento (1), the memento passes itself back into the list (2), which requests the link stored in the memento (3), which can then be queried to find the next link.) This is exactly the kind of access supported by owners-as-accessors: the collection owns its links, and only the collection can access them, but a memento can hold an incoming reference, but can never use that reference. As Gamma et al. say, using the Memento pattern rather than the Iterator pattern "*doesn't require breaking a collection's encapsulation to support iteration*" [21].

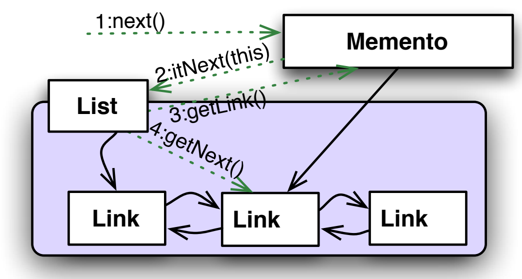

Fig. 2. Iteration by Memento, from [39]

We wished to adapt existing Java code, so we also implemented a second version of this design that kept an entire old Java-style structure-sharing iterator logically inside the collection instance, and used an independent proxy for that iterator as the memento outside (see fig. 3: when the next element is requested from the proxy (1), the proxy passes its incoming pointer to the iterator into the list (2), and the list requests the next element from the iterator (3), which returns that element as usual).

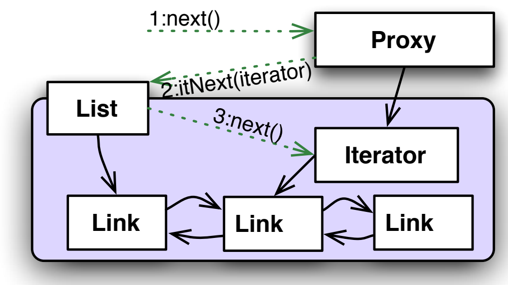

Fig. 3. List Iteration by Proxy, from [39]

We also wanted to compare the performance overhead imposed by dynamic ownership tests, especially when running on a modern JVM (Hotspot). To approximate this overhead, we added a dynamic check that compared an "owner" field added to the Memento and Proxy objects with the collection to which they belong. That is, in the dynamically checked versions of fig. 2 and fig. 3, when the list receives the itNext (...) call-back from the memento or proxy (2), the method checks that the memento or proxy's owner field refers to the list itself.

Fig. 4 presents the results for the IteratorLoops microbenchmark, showing that owners-as-dominators ("OasD") typically suffers a huge performance penalty over unmodified code ("Original"), but that the two owners-as-accessors designs ("Memento" and "Proxy") perform similarly to unmodified Java[1]. This microbenchmark also reveals little if any overhead of the dynamic check (conditions "Memento D." and "Proxy D.") as presumably the VM is able to remove the test via inlining in many cases — this is not particularly surprising as the code for the call-back in the memento is basically:

```
public E next() {
    return owner().iteratorNext(this);
}
```

while the code to implement iteratorNext in the collection is basically:

```
public E iteratorNext(Iterator<E> i) {
    if (i.owner() != this) throw new Error("Ownership Error!");
    return i.getLink().getNext();
}
```

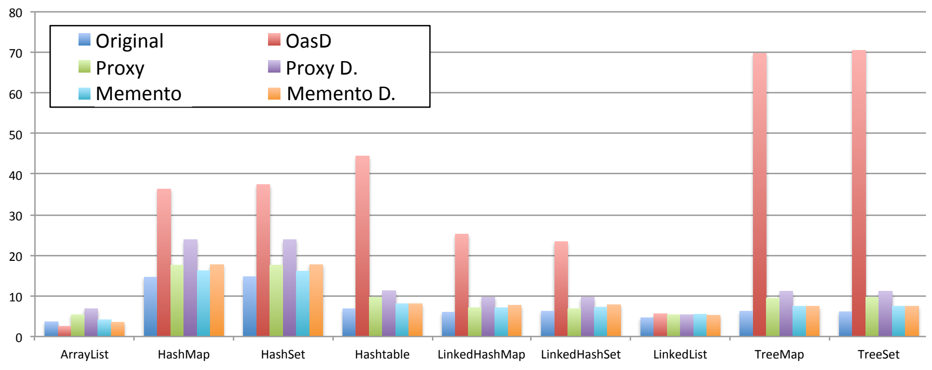

Fig. 4. Microbenchmark 1: IteratorLoops (single iteration time, *ns*) from [39]

We gathered similar (but more voluminous) data for the DaCapo, SPECjbb and SPECvm98 benchmarks. In our earlier work [39] we describe how, using the Asymptotic p-value of the Mann-Whitney U test (because the data are not normally distributed)

[1] Cognoscenti will notice that the LinkedList OasD benchmark has roughy the same performance as the other LinkedList benchmarks. This is because we implemented a one-place cache that optimised a single iteration through the list. The full paper illustrates that performance indeed degrades rapidly whenever that cache is missed [39], as is the case with all the other collections except ArrayList.

we tried to determine whether we could reject the null hypothesis that the mean time taken for the standard and refactored collections was the same. We were unable to show a statistically significant difference for all but 40 of 165 benchmarks. For this paper, we re-analysed the data (again using Mann-Whitney U tests) to see if we could reject the hypothesis that any of the other five conditions (owners-as-dominators, and memento and proxy with and without dynamic tests) were different to standard Java, for the three benchmark sets overall (Da Capo, SPECjbb2005, SPECjvm98). The means and medians are reported in the appendix — we performed pairwise tests for a selection of interesting cases and we could only reject the null hypothesis for e.g. SPECjbb: OasD vs memento and proxy (both $U = 0.000$, $p < 0.0005$) and dynamic memento vs proxy ($U = 210.5$, $p = 0.048$). Essentially what these results show is that owners-as-dominators often performs worse that any other design, but that on a current JVM owners-as-accessors can perform (nearly) as well as Java's designs based on incoming references — or at least that these standard benchmarks are unable to identify any difference. (More results, plus the full experimental protocol description, are in the full paper [39] and accompanying technical report [38].)

4 Objects + Owners = ADTS?

From rather practical questions about owners-as-accessors, we now turn to philosophical speculations.

In his landmark 2009 essay, William Cook compares and contrasts "two forms of data abstraction, *abstract data types* and *objects*" (his emphasis) [16]. Cook states that while the 'typical response is a variant of "objects are a kind of abstract data type" ' he considers that "neither one is a variation of the other". Jonathan Aldrich has also joined the discussion, arguing that "objects are inevitable" because they provide abstract interfaces to potentially multiple implementations, while abstract data types merely abstract a single implementation of an interface [1].

Based on facilities originating in Alphard [48] and CLU [29] Cook defines an ADT as consisting of "a public name, a hidden representation, and operations to create, combine and observe values of the abstraction". The identification of a "public name" emphasizes the fact that ADTs are not first class — certainly ADTs are not first class in most subsequent modular programming languages [47, 10, 31, 46]. An ADT has a hidden representation: this representation is not of the (non-first-class, singleton) ADT itself, but of the *instances* of the ADT — the values of the abstraction that are manipulated by the ADT's operations. Here, perhaps is the source of the temptation (or confusion) for designers of class-based languages as Cook describes: surely instances of a class are also instances of a corresponding ADT.

Owners-as-accessors admits another approach. First, we model the ADT itself as a first-class object. If we defined that object by a class, then we can create different "instances" of the whole ADT by creating different instances of the class (e.g. for type-genericity, or for configuration with different tuning parameters or algorithms). Then, the actual values of the ADT are represented by separate objects: probably (but not necessarily) declared within the ADT's class, but certainly owned by that class with an owners-as-accessors discipline. Operations on the ADT are written very similarly to the way they are written in CLU or other modular languages: as requests to the ADT instance, passing along the externally-opaque representation objects. Owners-as-accessors protects those representation objects from any attempts to read or modify them from outside the ADT object that owns them — but still allows the ADT's clients to store and compute with them just as with other ADT implementations.

```
class intSetADT.new {
 // representation
 type Cvt = {
  isEmpty −> Boolean
  insert(i  : Number) −> Number
  contains(i : Number) −> Number
 }
 method pair(car, cdr) −> Cvt is confidential {
   object is owned {
     def isEmpty is public = false
     method insert(i : Number) {
         if (contains(i)) then { self } else { pair(i, self) }}
     method contains(i : Number) {
         if (i == car) then { true } else { cdr.contains(i) }}
 }}
 def emptyset : Cvt is confidential = object is owned {
   method isEmpty −> Boolean {true}
   method insert(i : Number) −> Cvt { return pair(i, self) }
   method contains(i : Number) −> Boolean {false}
 }

 //ADT interface
 method isEmpty(s : Cvt) −> Boolean {s.isEmpty}
 method insert(s : Cvt, i : Number) −> Cvt {s.insert(i)}
 method contains(s : Cvt, i : Number) −> Cvt {s.contains(i)}
 method empty −> Cvt {emptyset}
}
```

Fig. 5. Integer Set ADT as an Object with Owners-as-Accessors

Figure 5 gives a Grace example of an owners-as-dominators style integer set ADT, after the examples of Cook [16] (originally from the CLU reference manual [27]). Following CLU, we define a type Cvt for the opaque representation objects, which are either pairs or the singleton emptyset. The **is** owned annotation[2] ensures that these objects are owned by the surrounding ADT object. Finally we implement the actual operations of the ADT itself in terms of the representation objects. These are simple because the representation is implemented as objects with methods; equally, we could have used e.g. plain data structures and typecases in the ADT methods.

[2] not currently part of the Grace implementation

This model does have the disadvantage that ADT operations will be dynamically dispatched when requested of the ADT object — but if a pure ADT programming style is used, then any references to the ADT will be statically bound and can be optimised very simply (as our study shows in the previous section). Ironically, CLU's implementation team considered OO-style dynamic dispatch to invoke ADT operations, "*In the early design of CLU, we thought that this kind of dynamic checking would be needed in some cases*" but abandoned this possibility in favour of a completely static design [28]. First class ADT objects of course have many advantages over static ADT modules: they can serve as Aldrich's dynamically substitutable service abstractions; they also can be defined incrementally by inheritance, simulated by other (ADT) objects, and share the other advantages Cook claims for objects.

From the perspective of owners-as-accessors, ADT and objects are indeed variations of each other — or rather, a classical ADT is a degenerate (singleton, statically bound) object which owns its instances. More generally, *an abstract data type is a kind of object that uses owners-as-accessors*. Owners-as-accessors is the only classic ownership discipline that facilitates this relationship. Owners-as-dominators is too strong, as the ADT's instances cannot pass out across the strong encapsulation boundary, and owners-as-modifiers is too weak, because it does not provide any encapsulation for the ADT instance objects.

5 Discussion and Conclusion

In this paper we have provided the first succinct definition of the owners-as-accessors ownership discipline. We have identified several existing ownership systems that have adopted this discipline. We re-analysed earlier performance results, and were again unable to show that iteration based on owners-as-accessors Mementos perform significantly worse than standard Java, while we could show owners-as-dominators iteration was often significantly slower. Finally we discussed how the opaque mementoes or "magic cookie" objects supported by owners-as-accessors have the same relationship to their owning object as instances of an ADT to the ADT itself.

It is important to realise that owners-as-accessors is a policy, not a mechanism; a specification, not an implementation. Like many other ownership conditions, owners-as-accessors can be enforced by static, dynamic, or hybrid checks; can rely on explicit annotations on programs; or be enforced against implicit rules; and can be discovered in existing programs by static or dynamic ownership inference.

Owners-as-accessors seems like a fruitful direction for further work. Quite some work in object-capability protection may be amenable to recasting in an owners-as-accessors framework. For example, Miller et al.'s escrow example tests whether two purse objects belong to the same mint object [32] by comparing the object identities of function closures, and van Cutsem has shown how membranes can be used to encapsulate whole object graphs, eventually relying on an on owners-as-dominators property to facilitate garbage collection [17].

Generalising the definition at the end of section 2 from "no access" to "*no **unmediated** access ...*" could also encompass work like Wernli et al. [45] into a "generalised owners-as-accessors" framework, where incoming references using an object are mediated in some way, rather than strictly forbidden. Treating a field assignment "o.f:=r" as inducing a *reference effect* on r (conceptually the effect of increasing r's reference count, but treated similarly to an attempted method invocation from o to r) could potentially incorporate even owners-as-dominators into the same framework. Finally, owners-as-accessors — with or without generalisations — offers a novel perspective on the relationship between objects and abstract data types which may be not only interesting in theory but even useful in practice.

Acknowledgements

Thanks to Monique Damitio for running benchmarks and Dalice Sim for statistical analysis, and to Stephanie Balzer and the anonymous reviewers for their comments. This work is partially supported by the Royal Society of New Zealand Marsden Fund.

References

1. Aldrich, J.: The power of interoperability: why objects are inevitable. In: Onward! pp. 101–116 (2013)
2. Aldrich, J., Chambers, C.: Ownership Domains: Separating Aliasing Policy from Mechanism. In: European Conference on Object Oriented Programming (ECOOP) (2004)
3. Balzer, S., Gross, T.R.: Verifying multi-object invariants with relationships. In: ECOOP Proceedings (2011)
4. Balzer, S., Gross, T.R., Müller, P.: Selective ownership: Combining object and type hierarchies for flexible sharing. In: FOOL (2012)
5. Bocchino, Jr., R.L., Adve, V.S., Dig, D., Adve, S.V., Heumann, S., Komuravelli, R., Overbey, J., Simmons, P., Sung, H., Vakilian, M.: A Type and Effect System for Deterministic Parallel Java. In: Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) (2009)
6. Boyapati, C., Liskov, B., Shrira, L.: Ownership Types for Object Encapsulation. In: Principles of Programming Languages (POPL) (2003)
7. Boyapati, C., Liskov, B., Shrira, L.: Ownership Types for Object Encapsulation. In: POPL Proceedings. pp. 213–223. ACM Press, New Orleans, LA, USA (Jan 2003), invited talk by Barbara Liskov.
8. Cameron, N., Drossopoulou, S., Noble, J., Smith, M.: Multiple ownership. In: ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA). pp. 441–460. ACM, New York, NY, USA (2007)
9. Cameron, N.R., Noble, J., Wrigstad, T.: Tribal ownership. In: OOPSLA Proceedings. pp. 618–633 (2010)
10. Cardelli, L., Donahue, J., Glassman, L., Jordan, M., Kalsow, B., Nelson, G.: Modula-3 language definition. SIGPLAN Not. 27(8), 15–42 (Aug 1992), http://doi.acm.org/10.1145/142137.142141
11. Clarke, D., Drossopoulou, S., Noble, J.: The roles of owners. In: IWACO (2011)
12. Clarke, D., Östlund, J., Sergey, I., Wrigstad, T.: Ownership types: A survey. In: Aliasing in Object-Oriented Programming, pp. 15–58. Springer-Verlag (2013)
13. Clarke, D., Potter, J., Noble, J.: Ownership types for flexible alias protection. In: OOPSLA (1998)
14. Clarke, D., Wrigstad, T.: External Uniqueness is Unique Enough. In: European Conference on Object Oriented Programming (ECOOP) (2003)

15. Clarke, D.: Object Ownership and Containment. Ph.D. thesis, School of Computer Science and Engineering, The University of New South Wales, Sydney, Australia (2001)
16. Cook, W.R.: On understanding data abstraction, revisited. In: OOPSLA Proceedings. pp. 557–572 (2009)
17. van Cutsem, T.: Membranes in javascript. `http://soft.vub.ac.be/ tvcutsem/-invokedynamic/js-membranes` (Mar 2012)
18. Dietl, W., Drossopoulou, S., Müller, P.: Separating ownership topology and encapsulation with generic universe types. ACM Trans. Program. Lang. Syst. 33(6), 20 (2011)
19. Dimoulas, C., Findler, R.B., Flanagan, C., Felleisen, M.: Correct blame for contracts: no more scapegoating. In: POPL. pp. 215–226 (2011)
20. Dimoulas, C., Moore, S., Askarov, A., Chong, S.: Declarative policies for capability control. In: Proceedings of the 27th IEEE Computer Security Foundations Symposium (Jun 2014)
21. Gamma, E., Helm, R., Johnson, R.E., Vlissides, J.: Design Patterns. Addison-Wesley (1994)
22. Gordon, C.S., Parkinson, M.J., Parsons, J., Bromfield, A., Duffy, J.: Uniqueness and reference immutability for safe parallelism. SIGPLAN Not. 47(10), 21–40 (Oct 2012), http://doi.acm.org/10.1145/2398857.2384619
23. Gordon, D., Noble, J.: Dynamic ownership in a dynamic language. In: DLS Proceedings. pp. 9–16 (2007)
24. Gruber, O., Boyer, F.: Ownership-based isolation for concurrent actors on multi-core machines. In: ECOOP. pp. 281–301 (2013)
25. Haller, P., Odersky, M.: Capabilities for uniqueness and borrowing. In: ECOOP Proceedings. pp. 354–378 (2010)
26. Lin, E., Rajan, H.: Panini: a capsule-oriented programming language for implicitly concurrent program design. In: SPLASH (Companion Volume). pp. 19–20 (2013)
27. Liskov, B., Moss, E., Snyder, A., Atkinson, R., Schaffert, J.C., Bloom, T., Scheifler, R.: CLU reference manual. Springer-Verlag (1984)
28. Liskov, B.: A history of CLU. In: History of Programming Langauges II. Addison-Wesley (1993)
29. Liskov, B., Snyder, A., Atkinson, R., Schaffert, C.: Abstraction mechanisms in CLU. Comm. ACM 20(8), 564–576 (Aug 1977)
30. Lu, Y., Potter, J.: Protecting representation with effect encapsulation. In: POPL Proceedings (2006)
31. MacQueen, D.: Modules for Standard ML. In: LISP and Functional Programming. pp. 198–207. ACM (1984), http://doi.acm.org/10.1145/800055.802036
32. Miller, M.S., Cutsem, T.V., Tulloh, B.: Distributed electronic rights in JavaScript. In: ESOP (2013)
33. Müller, P., Poetzsch-Heffter, A.: Universes: A Type System for Controlling Representation Exposure. In: Programming Languages and Fundamentals of Programming (1999)
34. Noble, J.: Iterators and encapsulation. In: TOOLS Europe (2000)
35. Noble, J., Clarke, D., Potter, J.: Object ownership for dynamic alias protection. In: TOOLS Pacific 32 (1999)
36. Noble, J., Vitek, J., Potter, J.: Flexible Alias Protection. In: European Conference on Object Oriented Programming (ECOOP) (1998)
37. Östlund, J., Wrigstad, T.: Multiple aggregate entry points for ownership types. In: ECOOP. pp. 156–180 (2012)
38. Potanin, A., Damitio, M., Noble, J.: Ownership and collections: The statistical analysis. Tech. Rep. ECSTR12-22, ECS, VUW, http://ecs.victoria.ac.nz/Main/TechnicalReportSeries (2012)
39. Potanin, A., Damitio, M., Noble, J.: Are your incoming aliases really necessary? counting the cost of object ownership. In: International Conference on Software Engineering (ICSE) (2013)

40. Potter, J., Noble, J., Clarke, D.: The Ins and Outs of Objects. In: Australian Software Engineering Conference (ASEC) (1998)
41. Rust Team: The Rust reference manual. Tech. rep., Mozilla Corporation (2014), from `rust-lang.org`
42. Shalit, A.: The Dylan reference manual: the definitive guide to the new object-oriented dynamic language. Apple Computer, Inc. (1996)
43. Skalka, C., Smith, S.: Static use-based object confinement. Springer International Journal of Information Security (2003)
44. Srinivasan, S., Mycroft, A.: Kilim: Isolation-typed actors for Java. In: ECOOP Proceedings (2007)
45. Wernli, E., Maerki, P., Nierstrasz, O.: Ownership, filters and crossing handlers. In: Dynamic Language Symposium (DLS) (2012)
46. Whitaker, W.A.: Ada - the project: The DoD high order language working group. In: HOPL Preprints. pp. 299–331 (1993)
47. Wirth, N.: Programming in Modula-2. Springer Verlag (1985), isbn 0-387-15078-1
48. Wulf, W.A., London, R.L., Shaw, M.: An introduction to the construction and verification of Alphard programs. IEEE Trans. Softw. Eng. SE-2(4), 253–265 (1976)

Appendix: Table of Results

Dacapo Times (ms) lower is better

	Mean	N	Std. Deviation	Median	Minimum	Maximum
Standard Java	20003.00	350	18783.548	18687.50	357	66853
Owners as Dominators	20038.92	350	18838.867	18901.00	369	68389
Proxy	20142.08	350	18860.387	19062.50	372	68000
Proxy Dyn.	20110.04	350	18873.821	18930.50	363	67498
Memento	20018.62	350	18703.768	19003.00	367	67864
Memento Dyn.	20137.31	350	18807.353	19694.00	373	68248
Total	20075.00	2100	18789.044	19622.00	357	68389

SPECjpp2005 bops (the higher the better)

	Mean	N	Std. Deviation	Median	Minimum	Maximum
Standard Java	29597.96	25	404.619	29557.00	28772	30279
Owners as Dominators	14061.68	25	180.795	14047.00	13785	14557
Proxy	28825.24	25	859.751	28724.00	27637	30690
Proxy Dyn.	28540.36	25	619.492	28561.00	27599	30141
Memento	28959.00	25	764.066	29017.00	27881	30879
Memento Dyn.	28393.68	25	640.569	28326.00	26824	29644
Total	26396.32	150	5581.431	28615.00	13785	30879

SPECJvm ops / m (the higher the better)

	Mean	N	Std. Deviation	Median	Minimum	Maximum
Standard Java	36.7534	380	26.05233	28.8800	5.52	109.13
Owners as Dominators	36.7458	380	26.13193	28.8500	5.81	108.57
Proxy	36.8576	380	26.18187	28.9500	5.67	108.94
Proxy Dyn.	36.7917	380	26.01006	28.9450	5.65	108.75
Memento	36.8182	380	26.26357	28.8750	5.71	109.08
Memento Dyn.	36.8045	380	26.08697	28.9450	5.81	108.87
Total	36.7952	2280	26.09261	28.9050	5.52	109.13