Object Ownership for Dynamic Alias Protection

James Noble
Object Technology Group
Microsoft Research Institute
Macquarie University
Sydney, Australia
kjx@mri.mq.edu.au

David Clarke, John Potter,
Practical Object Technology, Theory, and Engineering Research Group,
School of Computer Science and Engineering,
UNSW, Sydney, Australia.
clad@csee.unsw.edu.au,potter@cse.unsw.edu.au

Abstract

Interobject references in object-oriented programs allow arbitrary aliases between objects. By breaching objects' encapsulation boundaries, these aliases can make programs hard to understand and especially hard to debug. We propose using an explicit, run-time notion of object ownership to control aliases between objects in dynamically typed languages. Dynamically checking object ownership as a program runs ensures the program maintains the encapsulation topology intended by the programmer.

1: Introduction

Aliasing is endemic in object oriented programs — whether class-based or prototype-based [15]. Because objects have identity, and can be uniformly referred to by essentially any other object, the dependencies between objects in a program can be arbitrary.

Interobject references are particularly important because they can undermine programmer's attempts to build encapsulated aggregate objects — objects outside an aggregate object can hold references — *aliases* — to objects inside the aggregate, breaking the encapsulation of the aggregate object. Although many object-oriented languages include some kind of protection for the names of object's slots, so an internal sub-object can be stored in a *private* slot, name protection is insufficient to protect objects — by error or by oversight, a programmer can write a *public* method which returns a value from a *private* slot.

There have been a number of proposals to address this problem in classical object-oriented languages [10, 14, 2]. Most of these, including Islands [14] and Balloons [2], provide *Full Alias Encapsulation* — they statically restrict programs so that no aliases may cross an aggregate object's encapsulation boundary. Unfortunately, full alias encapsulation also forbids objects inside the aggregate from referring to other objects outside, preventing one object from being contained in two different collections.

We are developing a model for *Flexible Alias Protection* [21, 8] which separates the objects inside an aggregate into two categories — the aggregate's *representation*, which should not be accessible outside the aggregate, and the aggregate's *arguments*, which may be accessible outside, provided they are treated as immutable — similar to the distinction between aggregation and association in object-oriented design methodologies.

Flexible alias protection defines three invariants on a system's behaviour [21]:

- **No Representation Exposure** — An aggregate object's mutable representation objects should only be accessible via the aggregate object's interface.
- **No Argument Dependence** — An aggregate should not depend upon its arguments' mutable state.
- **No Role Confusion** — An aggregate should not return an object in one role when it was passed into the aggregate in another role.

Flexible alias protection is based on a model of object ownership. Every object which is part of the representation of an aggregate object is *owned* by that object, and should not be visible outside it — formally, an object's owner should be an articulation point (or dominator) on every path from the root of the system to the object. As a result, if an object is deleted or garbage collected, its representation — the objects it owns — can also be deleted. This notion of object ownership is quite general — for example, a UML object is related by *association* to its arguments and by *aggregation* to its representation [11] and a similar ownership relation between objects can be identified in all object-oriented programs [22].

For example, Figure 1 illustrates the structure of an object representing a university course. The Course object is represented by a List of Students, the List is represented by its Links, and the Students are the List object's arguments. The Course object is an articulation point for all the other objects, and the List object is an articulation point for the Links.

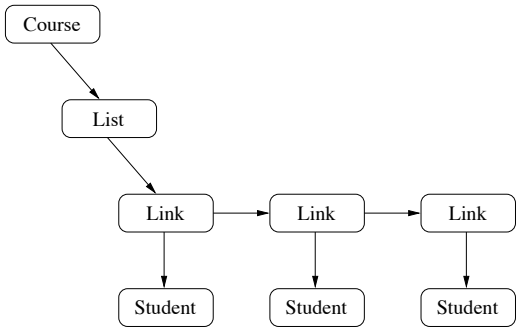

Figure 1. An aggregate course object

Flexible Alias Protection requires that any program manipulating the course structure maintains the three invariants (no representation exposure, no argument dependence, and no role confusion). For example, no representation exposure means that the Link objects may not be accessed outside their containing list (otherwise the List would not be an articulation point for the link objects), similarly, the Students should not be accessible to the wider system outside the Course object. No argument dependence means that the List object must not depend upon any mutable state contained with the Student objects. For example, if the List was required to be sorted in order of student's names, a student changing their name would render the list unsorted, and quite possibly break any methods inside the list object that depend upon the list

being sorted (such as a binary search or sorted list merge). No role confusion means that the List must not treat Link objects as if they were Student objects (or vice versa), even if they have the same type — as if, for example, the Student class inherited from the Link class.

2: Dynamic Alias Protection

Our existing models of flexible alias protection are classical — they are based on classes and static typing [21, 8, 9]. In this paper, we propose an explicit, run-time model for flexible alias protection based directly on object ownership and suitable for a dynamically-typed, prototype-based language such as Self [1] or Kevo [24].

One of the aims of prototype-based programming (compared with class-based programming) is to make objects *self-sufficient* — that is, an object should be able to be understood as an isolated component of a system, without reference to the enclosing system of which it is a part [20]. For example, most prototype-based languages allow each individual object to define its own structure and behaviour without reference to some external class. If necessary, groups of similar objects can then be created by *cloning* (shallow copying) a *prototype* object. Unlike inheritance, instantiation, and classes, which require special programming language support, there is nothing special about the clone operation or the objects used as prototypes: any object (whether cloned or created *ab initio*) is structurally independent from every other object in the system.

Given their aim of self-sufficiency, it is a little surprising that most prototype-based languages have not addressed the problems caused by interobject aliasing. If an object's representation is exposed, or if it depends on its argument objects, then the object cannot be understood in isolation from its enclosing context, and is not really self-sufficient.

Our dynamic alias protection scheme aims to extend the idea of self-sufficiency to address interobject aliasing, and is structured as follows. The ownership of every object is established and stored in a special owner field in each object, in much the same an object's class is stored in a special type or class field. Then, this ownership information is used to check the validity of each message send. Dynamic alias protection is based on runtime checks of objects' owner fields in exactly the same way dynamic type checking is based on runtime checks of objects' type fields.

2.1: Object Ownership

We propose adding an owner pseudovariable to every object in the program. The owner field simply stores object's owner — that is, it holds a reference to some other object in the program. An object's owner may be set explicitly when it is created, may be left uninitialised (or free) to represent an object which has no owner, or may be initialised to the special value proto if the object is to be used as a prototype.

One object is distinguished as the root of the system (in Self, this would be the lobby). The owner fields in the system should be constrained to form a forest, with trees rooted at the lobby and at free or proto objects, with each owner field pointing in the direction of the root of its tree. Intuitively, the objects making up an aggregate's representation will be owned by the aggregate.

Figure 2 shows the ownership structure for the Course example. The Course object is the owner of the List object, and the List is the owner of all the Links (although some links are reachable only via other Links). Similarly the Course object is the owner of all the Student objects, even though the Students are only reachable from the Course via the List and Links.

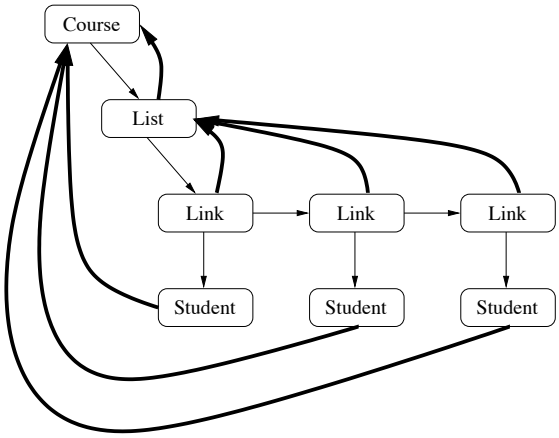

Figure 2. The ownership structure of the course object

2.1.1: Object Pathnames

Object ownership can be also modeled as giving *pathnames* to objects, similar to the pathnames in a file system, except that pathnames do not necessarily uniquely name objects. Each object is (conventionally) given a local name — perhaps based on the name of its type or class — and an object's pathname is the concatenation of all its owners' local names, and then its own local name. The lobby is implicitly at the start of every path, so simple paths like a and b mean that a and b are in the lobby. A path such as a.b.c refers to an object c owned by an object b in turn owned by another object a which is within the lobby.

To simplify the presentation we will assume that each object is uniquely named by a pathname. Objects' pathnames will be unique only if object's local names are unique within their owners — in a practical system this could be arranged simply by using objects' underlying identifiers (such as memory pointers) as their local names. To continue the course example from Figure 2, if the main course object has the pathname course, then the list has the pathname course.list, students s1 and s2 would have pathnames course.s1 and course.s2, and the links inside the list would be course.list.link1, course.list.link2, and so on.

2.1.2: Ownership and Containment

Pathnames can make the concept of object ownership clearer: an object's owner has the same pathname as that object with the last element (the object's local name) removed. Some object o owns another object r (that is, r is part of o's representation) if r's pathname is o's pathname extended with r's local name l, i.e:

$$o \textbf{ owns } r \quad \equiv \quad o.l = r \quad \equiv \quad \mathsf{owner}(r) = o$$

Assuming we didn't know the ownership structure shown in Figure 2, we could deduce it from the object's pathnames: course.list is the owner of course.list.link1 and course.list.link2 because it is the direct prefix of their pathnames.

Pathnames also illuminate object *containment*. Containment is the transitive closure of object ownership: one object is contained within another object if the containing object is the contained objects owner, the owner's owner, and so on. In terms of pathnames, one object is

contained within another object if the container object's pathname is any prefix of the contained object's pathnames. That is:

$$o \textbf{ contains } r \equiv r \text{ is a prefix of } o$$

In the example in Figure 2, the list contains the links (because it owns them); the course contains the list and the students (because it owns them); the course also contains the links (because they are contained in the list, which the course owns); and the lobby contains everything.

2.2: Alias Protection Rules

Dynamic alias protection uses object ownership to enforce the invariants for flexible alias protection. Aliasing problems occur when the internal state of an aggregate object is changed via an alias, without going through the interface of that object. That is, when an aggregate's state is changed from the outside (representation exposure) or when an aggregate depends upon the mutable state of an external object (argument dependence).

We can use object ownership to forbid such accesses, by imposing two rules onto the system, the representation rule and the arguments rule.

2.2.1: Representation Rule

To avoid representation exposure, we must prevent an object receiving a message which crosses into an object from the outside — in the example, Link objects may be accessed from the List, but not from the Course. To this end, we enforce a *representation rule* on all message sends in the system, which disallows messages which break encapsulation.

The representation rule states a *sender* object s can send a message to a *receiver* object r only if the receiver's owner contains the sender. That is:

$$s \rightarrow_{rep_1} r \Rightarrow \mathsf{owner}(r) \textbf{ contains } s$$

The effect of the reference rule is that the only message sends which will proceed are those which originate in the parts of the system contained within the receiver's owner. We call this set of objects the receiver object's *extent*.

$$s \in \mathsf{extent}(r) \Rightarrow s \rightarrow_{rep_1} r$$

One way to think about this is that messages may first go *up* any number of levels in the containment hierarchy (even up to the lobby) but then only ever go *down* one level to get to its implementation. Going down more than one level would mean that a message would penetrate through an object's encapsulation boundary.

Any message sends which do not meet the rule should raise a dynamic "Representation Rule Check" exception, terminating the message send in the same way that a message which fails a dynamic type test raises a "Does Not Understand" error in Smalltalk, or a dynamic cast raises a "Class Cast" exception in Java.

2.2.2: Argument Rule

The representation rule prevents representation exposure, but does not address argument dependence. Argument dependencies occur when some object depends on some information lying outside it — that is, which it does not contain.

The basic representation rule rep_1 as presented in the preceding section does not distinguish between messages sent to an object's arguments, and messages sent to an object's representation. Under the representation rule, a message may pass from the sender to some argument object outside it, and that argument object can be modified independently of the sender.

Argument dependence can only occur when a message is passed out of an object, that it is, when it is sent to some object it doesn't contain. A message sent to an object contained in the receiver cannot therefore cause argument dependence:

$$s \rightarrow_{noarg} r \Rightarrow s \textbf{ contains } r$$

Combining this rule with the basic representation rule (rep_1) produces a revised representation rule (rep) which prevents argument dependence as well as representation exposure:

$$s \rightarrow_{rep} r \Rightarrow s \textbf{ owns } r$$

(since $\mathsf{owner}(r)$ **contains** $s \wedge s$ **contains** $r \Rightarrow s$ **owns** r).

Unfortunately, the revised *rep* rule is very restrictive; it only allows an object to send a message to its own direct representation. This prevents all message sends to objects' arguments, even though many messages sent to arguments do not cause argument dependence. Argument dependence can only arise when information which can change flows into an object from outside — that is, when an object depends upon the *mutable state* of its arguments.

For this reason, we introduce the *argument rule* to permit sending messages to arguments, whenever those messages cannot cause argument dependence. The argument rule requires messages sent to an object's arguments to fit in to one of two categories.

Messages sent to arguments must either:

- never access mutable state

or

- never return any information to the sender.

We call messages that access mutable state *immutable* messages, and messages that never return any information to the sender *one way* messages. Where necessary, we use the term *mutable* messages to describe "normal" messages that may access mutable state and return information to the sender.

The argument rule applies whenever the primary representation rule (rep_1) applies, but the revised rule (rep) does not:

$$s \rightarrow_{arg} r \Rightarrow \mathsf{owner}(r) \textbf{ contains } s \wedge \textbf{ not } s \textbf{ owns } r$$

Finally, in the dynamic alias protection system, all messages must be covered by one or other of the *rep* or *arg* rules, or an alias protection exception is raised. In Figure 2, for example, the

List can send mutable messages to its Links, and the Course can send mutable messages to its Students. The List or Links cannot send mutable messages to the Students (this would make the List depend on its arguments), but can send immutable or one way messages.

The asymmetry between the representation and argument rules is important. The representation rule is defined in terms of direct ownership, while the argument rule is defined in terms of transitive containment. This is to protect the integrity of internal implementation objects — if the representation rule used containment, an object's representation would be able to be exposed because a message send could go arbitrarily deep into an object's representation (in Figure 2, a Course sending a message to a Link would expose the List's representation).

2.2.3: Immutable Messages

Operationally, immutable messages can be naïvely implemented by a flag which keeps track of *mutable sends* versus *immutable sends*. Mutable messages may access mutable state — read and write objects' variables — but if an immutable send attempts to access mutable state, a runtime "Mutable State Exception" should be raised. A message send is mutable by default, but once a thread begins executing an immutable message, all messages in that thread will be immutable until the original immutable message returns.

In order for this rule to be practical, a programming language needs to be able to express idempotent computations easily. For example, the language could incorporate purely functional constant expressions and single-assignment (once) slots in objects (as in Cecil [7]).

2.2.4: One Way Messages

In a one way message send, no information may flow back from the receiver to the sender of the message. This means that one way messages may not return results — or, rather, that if the methods called by the messages return results, that these results should not be delivered to the message sender. Any other channels through which information could flow back to the receiver must also be disabled; in particular, if the receiver raises an exception processing the message, the exception cannot be communicated back to the sender.

The advantage of one way messages is that they can read and write mutable state in the receiver, so they allow information to flow out of objects into their arguments. For example, assuming all windows in a GUI library are owned by the lobby, then any object can send a one-way message to update the display of any window. The Observer pattern typically uses this kind of design.

2.2.5: Role Confusion

Role confusion is represented as type parameterisation in static alias protection systems [21, 8]. The problem in these static systems is that containers must somehow record the ownership types of the objects they contain, since ownership cannot be checked dynamically when objects are removed from containers. For the dynamic alias protection, every object records its ownership in the owner field, and any ownership violations will be detected through the dynamic rules, including those that would result in role confusion in a static system.

3: Discussion

Dynamic alias protection and object ownership interact with other language features, in particular creating new aggregate objects by cloning, and the prototype corruption problem.

We also discuss a variety of possible extensions to dynamic alias protection.

3.1: Clone

Existing clone primitives are either *shallow* — copying only one object, or *deep* — recursively copying an object and all objects to which it refers, ideally duplicating each copied object only once. Neither of these clones produces a useful copy of an aggregate object — a shallow clone copies only the top level object, while a deep clone will copy an aggregate plus any argument objects which do not belong inside the aggregate but to which the aggregate refers (assuming the deep clone does not attempt to clone every object in the system).

Object ownership can be used to guide the actions of the clone primitive, cloning an aggregate object and its representation while not cloning references to the aggregate's arguments. That is, cloning all objects owned (transitively) by the object being cloned, while keeping references to other (necessarily external) objects intact. This is somewhere between a shallow clone and a deep clone[1].

Operationally, an ownership-based clone operation must take account of two issues. First, when the clone reaches a candidate object to be cloned, it must traverse up from the candidate object's owner field to see if it is owned (possibly transitively) by the object actually being cloned. If so, the candidate object is part of the cloned object's representation (or the representation of some object which is transitively part of the cloned object) and so should be cloned; otherwise, it is an argument object and so should not be cloned. Second, the clone operation must be aware that the structure of the objects it is cloning will be a graph, rather than a tree, so it must take care to clone any object only once, typically by maintaining a map from every object to their clone [23, p.314]. Finally, the owner fields in the new aggregate object must be initialised to mirror that of the original aggregate, and the owner field in the new aggregate must be set to free.

Figure 3 shows the result of cloning the List inside the Course from Figure 2. The List and its representation Links are cloned, but the new List shares its argument Students with the original List. In contrast, Figure 4 shows the result of cloning the whole Course — the entire Course is replicated, including its constituent List, Links, Students, and ownership structure.

3.2: Prototype Corruption

Object ownership addresses the prototype corruption problem [4]. This problem arises when a prototype is accidently modified by a program, typically because the prototype was not cloned before being used, and so the prototype (rather than a clone of the prototype) is directly incorporated into the running program.

We set a prototype's owner field to the special value proto, which marks the object as a prototype. The value proto can be though of as representing an object which owns the root of the system (the lobby). Since prototypes are not owned by any other objects in the system, the arguments rule prevents them from being corrupted by any other objects. Unfortunately, since cloning an object requires access to the object's mutable state, a strict interpretation of the mutability rule would prevent prototypes being cloned. The mutability rule is therefore relaxed for object's whose owner is proto; these objects may be cloned but not otherwise accessed. Because prototypes are owned above the root of the system, their extent is the whole system.

[1] We are considering calling this a *sheep clone* after Dolly [6].

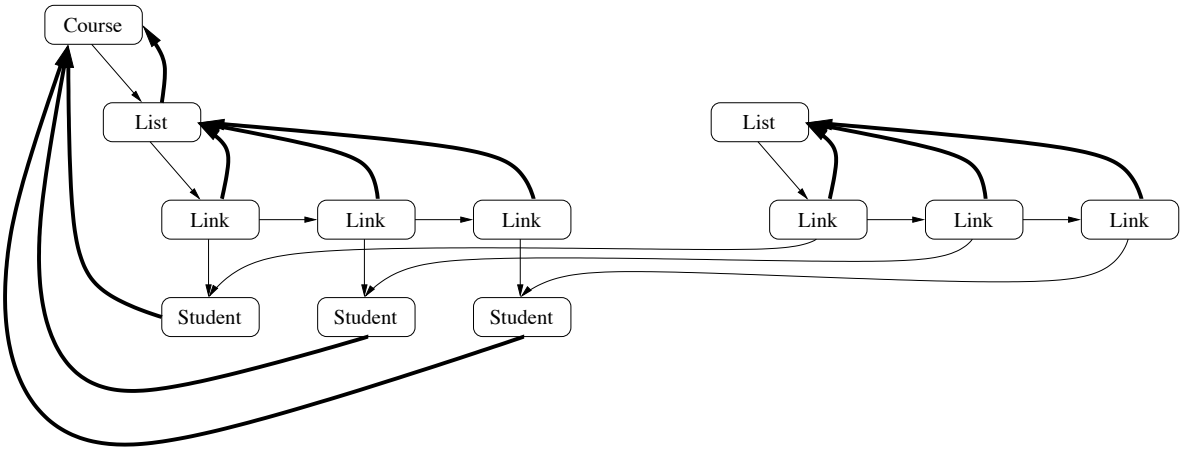

Figure 3. Cloning the list

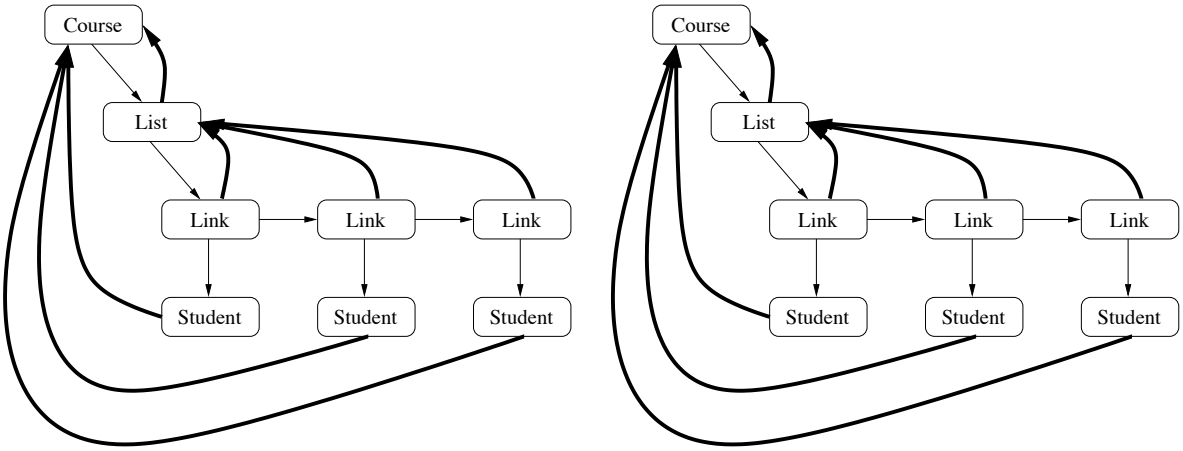

Figure 4. Cloning the course

3.3: Sibling Rule

We are considering a number of extensions to the ownership scheme. The representation rule limits collaboration between subsidiary objects implementing an aggregate object — indeed, since they will both be owned by the aggregate, they cannot access each other's mutable state; in fact, they cannot even access their *own* mutable state through a double dispatch via the other object. For example, the mutable state of the link objects in Figure 2 can only be accessed by the list object which owns them; one link object cannot change another link object. The core of this problem is that these kind of *auxiliary objects* do not represent abstractions in their own right; they are only used to implement other abstractions. For auxiliary objects, the representation rule can be weakened, so that auxiliary objects can access the mutable state of other auxiliary objects, provided they are owned by the same object. We call this the *sibling rule*, since the auxiliary objects are all owned by the same parent object.

$$s \rightarrow_{\mathit{sibling}} r \Rightarrow \mathsf{owner}(r) = \mathsf{owner}(s)$$

3.4: Reference Containment

The representation rule prevents the mutable state of an object's representation from begin accessed outside that object, and avoids all the problems of representation exposure. It is interesting that, technically, the representation rule does not prevent representation exposure, because any object may keep references to an object belonging to another object's representation. What the representation rule does ensure is that no messages can ever be sent through such an exposing reference, so the "exposure" has no practical effect. In this sense, neither the representation or argument rules actually control aliasing, because they apply to message sends rather than pointers. Rather, they prevent it causing problems for programmers.

As with many forms of dynamic typing, what this means in practice is that errors will often only be detected long after the situation that caused them has passed. Part of some obejct's representation can be handed out of that object, but the error will only be detected when another object sends the exposed representation object a message.

The key point here is that the only objects which may legitimately refer an object are objects which can send that object a message — the object's owner or other objects transitively owned by its owner, that is, the object's extent, as defined by the basic representation rule. This ensures that an object's owner is an articulation point on every path to the object [22].

The *reference rule* enforces this condition:

$$s \rightarrow_{points\ to} r \Rightarrow s \in \mathsf{extent}(r)$$

The reference rule checks, for each non-self send, that when a message is sent the *receiver* is within the extent of every argument passed into the message, and that when a message returns that the *sender* is within the extent of the message's return value.

Operationally, this rule could be checked by inspecting the owner field of the arguments and return value of every non-self message send, and comparing it with the chain of owner fields starting from the message's receiver (for arguments) or sender (for results) and following it up to the lobby. The owner of the argument (or result) should be somewhere within the receiver's (or sender's) owner chain.

3.5: Exporting References

The sibling rule does not cover all cases. An external iterator [12], for example, may need to access the internal representation of the object over which it is iterating. An iterator cannot be an auxiliary object, because auxiliary objects cannot access each other's representations directly, but can only send each other messages. This problem can be addressed by allowing objects to *export* other objects, granting the exported objects access to their representation and allowing them to send messages to any objects they own. For example, after an export operation, an iterator would be placed on its aggregate object's *access control list*. The mutability and reference rules could be extended so that objects are considered to be owned by the object in their owner field, and any exported objects in that object's access control list.

4: Related Work

A number of prototype-based languages have included some support for object ownership along with basic references and inheritance. Amulet and Garnet allow objects to have parts

as well as slots; when an object is cloned, all its parts are also cloned [18, 19]. ThingLab [5] also built objects from composition hierarchies, although it used paths (rather than references) to access internal components of aggregate objects and similar schemes have been suggested for Smalltalk [3]. Object ownership has also been proposed for Eiffel [16], by annotating fields as either *private*, to hold representation objects, or *protected*, to hold objects belonging to the representation of the owner of the object. Similarly, the Dee language informally marks variables to show whether they refer to the inside or outside of an aggregate [13]. Other aliasing control mechanisms for object-oriented languages implicitly introduce some notion of object ownership. For example, an Island's Bridge class [14] or a Balloon Type [2] effectively owns the objects contained within the scope of their Island or Balloon respectively.

It is also interesting to compare the dynamic alias protection scheme (presented in this paper) with our static flexible alias protection scheme [21]. Flexible alias protection and dynamic alias protection aim to provide the same guarantees about programs' aliasing behaviour, but use totally different mechanisms to provide those guarantees. These two schemes illustrate the classic trade-offs between compile-time and run-time checking — flexible alias protection uses annotations onto static types (called modes) that are checked at compile-time, while dynamic alias protection uses annotations onto objects (the owner fields) that are checked at run-time. The static scheme is more restrictive — for example, it forbids containers of objects with heterogeneous ownership — but it does not impose any execution overhead, unlike the dynamic scheme. An explicit rule against role confusion rule is not required in the dynamic scheme, as every object knows its owner, and passing an object into or out of an aggregate cannot change that object's ownership.

Unlike much related work [14, 2, 17], to date, neither our dynamic nor static alias protection schemes rely on uniqueness — although we forbid mutable message sends and references between certain objects, if references are permitted, the number of references is not constrained.

5: Conclusion

A great advantage of prototype-based languages is that they support independent, self-defining objects. Like objects in classical object-oriented languages, these objects can have dependencies on other objects, accessing those objects via inter-object references. We have described proposed a dynamic model for alias protection for prototype-based languages, based on an explicit notion of object ownership. Each object is given an extra owner field, and this field is used to maintain restrictions the messages objects can receive. By controlling the effects of aliasing, dynamic alias protection should make objects in prototype-based languages more self-sufficient, and thus make programs easier to understand and debug.

Acknowledgements

Thanks to the anonymous reviewers for their comments on this paper. This work was supported by Microsoft Australia Pty. Ltd. An earlier version of this paper was discussed at the OOPSLA'98 Workshop *Thinking with Prototypes*.

References

[1] Ole Agesen, Lars Bak, Craig Chambers, Bay-Wei Chang, Urs Hölzle, John Maloney, Randall B. Smith, David Ungar, and Mario Wolczko. *The Self Programmer's Reference Manual.* Sun Microsystems and Stanford University, 4.0 edition, 1995.

[2] Paulo Sérgio Almeida. Baloon Types: Controlling sharing of state in data types. In *ECOOP Proceedings*, June 1997.

[3] D. Blake and S. Cook. On including part hierarchies in object-oriented languages, with an implementation in Smalltalk. In *ECOOP Proceedings*, pages 41–50. Springer-Verlag, 1987.

[4] Günther Blaschek. *Object-Oriented Programming with Prototypes.* Springer-Verlag, New York, N.Y., 1994.

[5] Alan Borning. The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4), October 1981.

[6] K.H.S. Campbell, J. McWhir, W.A. Ritchie, and I. Wilmut. Sheep cloned by nuclear transfer from a cultured cell line. *Nature*, 380:64–66, March 1996.

[7] Craig Chambers. The Cecil language: Specificaion and rationale. Technical report, Department of Computer Science and Engineering, The University of Washington, 1997.

[8] David Clarke, John Potter, and James Noble. Ownership types for flexible alias protection. In *OOPSLA Proceedings*, 1998.

[9] David Clarke, Ryan Shelswell, John Potter, and James Noble. Object ownership to order. MRI Technical Report Submitted for Publication, 1998.

[10] David L. Detlefs, K. Rustan M. Leino, and Greg Nelson. Wrestling with rep exposure. Technical Report SRC Research Report 156, DEC Systems Research Center, july 1998.

[11] Martin Fowler and Kendall Scott. *UML Distilled: Applying the Standard Object Modeling Language.* Addison-Wesley, 1997.

[12] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns.* Addison-Wesley, 1994.

[13] Peter Grogono and Patrice Chalin. Copying, sharing, and aliasing. In *Proceedings of the Colloquium on Object Orientation in Data bases and Software Engineering (COODBSE'94)*, Montreal, Quebec, May 1994.

[14] John Hogg. Islands: Aliasing protection in object-oriented languages. In *OOPSLA Proceedings*, November 1991.

[15] John Hogg, Doug Lea, Alan Wills, Dennis deChampeaux, and Richard Holt. The Geneva convention on the treatment of object aliasing. *OOPS Messenger*, 3(2), April 1992.

[16] Stuart Kent and Ian Maung. Encapsulation and aggregation. In *TOOLS Pacific 18*, 1995.

[17] Naftaly Minsky. Towards alias-free pointers. In *ECOOP Proceedings*, July 1996.

[18] B. A. Myers, D. A. Guise, R. B. Dannenberg, B. Vander Zanden, D. S. Kosbie, E. Pervin, A. Mickish, and P. Marchal. Garnet: Comprehensive support for graphical, highly interactive user interfaces. *IEEE Computer*, 23(11), 1990.

[19] Brad A. Myers, Rich McDaniel, Rob Miller, Alan Ferrency, Patrick Doane, Andrew Faulring, Ellen Borison, Andy Mickish, and Alex Klimovitski. The Amulet environment: New models for effective user interface software development. Technical Report CMU-HCII-96-104, Human Computer Interaction Institute, Carnegie Mellon University, 1996.

[20] James Noble, Antero Taivalsaari, and Ivan Moore, editors. *Prototype-Based Programming: Concepts, Languages and Applications.* Springer-Verlag, 1999.

[21] James Noble, Jan Vitek, and John Potter. Flexible alias protection. In *ECOOP Proceedings*, 1998.

[22] John Potter, James Noble, and David Clarke. The ins and outs of objects. In *Australian Software Engineering Conference (ASWEC)*, 1998.

[23] Trygve Reenskaug. *Working with Objects: The OOram Software Engineering Method.* Manning Publications, 1996.

[24] Antero Taivalsaari. *A Critical View of Inheritance and Reusability in Object-oriented Programming.* PhD thesis, University of Jyväskylä, 1993.