Overcoming Representation Exposure

David Clarke, James Noble, John Potter
Microsoft Research Institute
Macquarie University, Sydney
clad@mri.mq.edu.au

Abstract

Through their reliance on reference semantics, what objects gain in flexibility they lose in their ability to control access to their internal implementation details — this is the problem of *representation exposure*. Existing approaches to achieving better encapsulation for objects, including type systems and programming models, are based on object encapsulation boundaries and constrained access across those boundaries.

Our earlier work dealing with ownership types achieved a limited form of representation containment, whereby each object's representation was contained within that object, its owner. Inter-object accessibility was constrained so that owners were dominators of the objects they owned. Although achieving a clear statically expressible notion of containment, our approach lacked flexibility, so that common programming techniques became overly cumbersome to implement.

This paper investigates new and improved ways for expressing object structure, using path variables, and separates the notion of ownership from containment, thereby providing greater flexibility. We illustrate the ideas through an extension of Abadi and Cardelli's object calculus, and demonstrate solutions to some problems of representation exposure.

1 Representation Exposure Tension

Aliasing is problematic because it breaks the encapsulation necessary for building reliable software components [16]. A common symptom of aliasing is representation exposure which occurs when the internal implementation of an aggregate object is accessible outside the aggregate [14]. A system controlling aliasing must firstly prevent representation exposure, but it must also be practical, flexible, and powerful; allowing both the implementation of efficient solutions and the building of complex abstractions. That is, the system must be expressive, while preventing representation exposure.

In earlier work we presented *ownership types* [6], which is based on labelling each object with its owner and recording this information in the object's type. Ownership is fixed for an object's lifetime and there is a distinction between being an *owner of* and having a *reference to* an object. This allows, for example, an object to be stored in a container even though they are not owned by the container. The evolving object graph of a well-typed program has the property, called *representation containment*, that all access paths from the root of the system to an object pass through that object's owner, making an object's owner a dominator for such paths [22]. Unfortunately, ownership types have limited expressiveness, not accomodating inheritance, and being unable to express the examples considered in this paper. Nevertheless, to our knowledge, ownership types constitute the first type system which enforced representation containment. By distilling ideas from that work with recent insight into Abadi and Cardelli's object calculus [1], we overcome these shortcomings.

Two predominant proposals for protecting aliasing which rely on restricting the scope of object visibility are Islands [13] and Balloons [2]. These systems require *full alias encapsulation* which forbids a reference from inside an encapsulation boundary to an external object. Extra operations, such as destructive read and copy assignment, are required to move objects across encapsulation boundaries. Minsky's u-pointers [18] rely on unique pointers and destructive reads, so ultimately this offers no more than Balloons. Eiffel's expanded types [17] and C++'s values and copy semantics [8] enforce only shallow containment, precluding linked data structures, and are inflexible. Flexible alias protection [20], from which this work derives, introduces the idea of *flexible alias encapsulation.* Access to objects outside the encapsulation boundary is permitted only through interfaces which do not depend on the mutable state of the external objects. Kent and Maung's [15] proposal is similar to ours. Their ownership checks occur only at run-time, whereas we do it statically.

Some formal approaches to the problem also exist. Detlef, et. al 's approach is specification based and it specifically attacks representation exposure [7]. They also present a collection of problems which systems dealing with representation exposure should handle which we address here. Hoare and Jifeng's trace-based model [12] deals with aliasing directly by recording paths of field names which denote the same object. The authors admit that their work has not been following to any conclusion. In the presence of subtyping, their model seems intractible.

Greenhouse and Boyland's Object-oriented Effects System provides a partition of the object space in a similar manner to ownership types, except that their *regions* are hierarchical but inclusive (subset ordering), whereas the partition induced by ownership types is hierarchical but exclusive.

Other related work is surveyed in our previous papers [6, 20, 22].

In this paper we present a number of examples in our calculus. We demonstrate how to build simple containers and more complex aggregates like iterators. We show how a method can be coded independently of its arguments' owners, and demonstrate how an object's representation can be created externally without any reference being retained to the intended representation. Sometimes the flexibility these approaches is not enough, and direct access to the representation is desired. We also demonstrate how to do this safely.

Overview. In Section 2 we discuss our model of representation containment. We informally present an extension to Abadi and Cardelli's object calculus in Section 3. A more complete treatment is in preparation. Section 4 presents our solutions to problems mentioned in the literature and ones we have encountered in our own research. We then list problems for which we provide no satisfactory solution in Section 5.

2 Representation Containment

In our previous work [6], we presented a type system for a class-based programming language (without inheritance) which enforced a property called *representation containment.* The types, called *ownership types*, express the owner of an object in the object's type. Every object has an owner which is another object or the ambient root object of the system.

We use *contexts* to denote owners. Contexts and the rules which govern them describe and limit the extent of object access. They are represented by paths π and path variable α specified as follows:

$$\begin{array}{rcl} p, q & ::= & \pi \mid \alpha \\ \pi & ::= & \epsilon \mid \pi.\kappa \end{array}$$

where κ is a denumerable collection of names. The empty path, ϵ, represents the root, and a path $\pi.\kappa$ is a context inside context π. The relation $\subset$ between contexts, called *inside*, is defined as follows:

$$p \subseteq q \quad \text{if and only if} \quad q \text{ is a prefix of } p$$

All contexts are inside the context ϵ, that is, for all p, we have $p \subseteq \epsilon$.

Each object has two associated contexts: the object's owner, and its representation context, the owner of its representation. The owner is used to determine which other objects can keep a reference to this object. The representation context is used to determine which objects this can keep a reference to. We specify the owner when creating a new object. The representation context, however, is a distinct new context. In our earlier work, if an object had owner π, the the representation context was $\pi.\kappa$, where κ was a fresh atom [6][1]. By identifying objects with their representation contexts, we see that objects are inside their owners.

We proved that if the following constraint is maintained for all objects α and β

$$\alpha \rightarrow \beta \Rightarrow rep(\alpha) \subseteq owner(\beta) \qquad (*)$$

where $\alpha \rightarrow \beta$ means α has a reference to β, $rep(\alpha)$ is α's representation context, and $owner(\beta)$ is β's owner, *then* every object's owner was a dominator for paths from the root of the system to that object, and that the tree of contexts was the dominator tree for the object graph [22]. This means that external access to an object's representation must be through an object's owner. We also proved that programs well-typed in the ownership type system satisfied the constraint $(*)$ [6].

The representation context contains protected objects which not accessible outside their owner. The objects potentially accessible to an object are those owned by an object outside its representation context, and no others. In particular, an object has access to objects which have the same owner. This allows linked data structures to be built, with each link having the same owner (see [6] for an example). The representation of an object need not only be the objects stored directly within fields. It can be an arbitrary object graph, with every node receiving the same protection.

Unfortunately our model is not expressive enough. To be practical, the we must allow certain coding styles, even when they are potentially dangerous, at least in current systems which offer no represention protection. As an example, due to Detlefs, Leino, and Nelson [7], consider a lexical analyser (lexer) built on top of a stream reader interface (reader). The reader is part of the lexer's representation; alternatively, the lexer owns the reader. For efficiency reasons, we want access to a read-only interface of the stream underlying a lexer from outside of the lexer, thus exposing representation. Also, for the lexer to be reusable, the reader must be created externally to the lexer, so that the lexer can be parameterized by different reader implementations. In this case, we have the constraint that no reference to the reader is retained outside the lexer.

We must also allow the building of complex aggregate objects, such as aggregates with more than one object acting as an interface to the aggregate. A linked list with an external iterator in the style of Java's `Enumeration` interface is one such example [10]. The links of the list constitute the representation. The main interface allows elements to be added or removed from the list. Auxiliary interfaces, the `Enumeration` objects, can iterate over the elements of the list. These iterators must access the representation of the list. The current version of our model does not allow the iterator example. The reason is that there is path from the root

[1] The presentation differs in [6], but the underlying semantics are the same as described here.

of the object graph which does not pass through the owner of the representation, violating (*). We describe the appropriate extension next, and deal with particular applications in the remainder of the paper.

We extend the model to allow multiple objects access to an object's representation, so we can build safe aggregates or components which present multiple interfaces, each of which is an object having its own state. This is achieved by relaxing constraints on where the representation context is created. Previously an object's representation is created directly inside the object's owner. We now allow the representation context to be created anywhere inside the owner of the object. In particular this allows us to create an object with the same owner as a given object, but with its representation context inside the first object's representation context. Because these two objects have the same owner, they are accessible by the same objects. But since the second object's representation context is inside the first object's, the second object has access to the first object's representation. We maintain the same invariant (*) with appropriately modified definitions of $rep()$ and $owner()$.

This looser requirement allows objects to be built inside other objects, have access to the representation, but be allowed themselves to pass through the protection boundary of the original object, in the same way that iterators do.

The system provides instance based protection. Each object has its own representation context distinct from all others including objects from the same class. This kind of protection is orthogonal to the selective export mechanisms such as Java's packages, C++'s friends, privacy in these languages, and Eiffel's export mechanism. These export controls do not statically guarantee representation containment, though they assist with achieving it in current programming languages. Our system can *guarantee* that representation is not exposed.

3 A Language with Representation Containment

We present an overview of a programming language core which expresses representation containment. It is an extension of Abadi and Cardelli's first-order object calculus [1] with a few restrictions on the kinds of expressions allowed, namely that functions are not first-class citizens and there is no method update. These are both disallowed because they can be used to break the containment invariant (*).

Abadi and Cardelli's objects have the form:

$$[l_i = \varsigma(x_i)b_i{}^{i\in 1..n}],$$

where l_i are the method labels, and $\varsigma(x_i)b_i$ is a method where x_i is the self parameter. When the method is invoked, the whole object is substituted for the self parameter into the body of the method. If the $\varsigma(x_i)$ part is missing, then the entry is a field. Method bodies are generally λ-abstractions. To prevent functions from being first-class, we require that λ-abstractions are fully applied and that no field contains a function. We disallow closures because the underlying type system cannot keep track of the contexts accessed within them. With first-class functions we lose representation containment.

Field assignment is allowed only valid when the type of the assignment source is compatible with the field's type. Types are compatible only when their owners and the owners of their fields, and so forth, are equal, and the types are compatible in the usual sense.

We extend these objects with annotations for owner and representation contexts as follows

$$[l_i = \varsigma(x_i)b_i{}^{i\in 1..n}]^p_q$$

where superscript p is the owner and subscript q is the representation. It will always be the case that $q \subset p$. The owner and representation contexts of an object will either be ϵ, the top level context, or one created using the **new** construct, which is described later. This kind of object is the internal view of an object, without any mechanism protecting the representation — it is not used directly. To denote an object which does not have an owner, we use the root context ϵ. This is equivalent to `norep` (*nobody's representation*) from our earlier work [6]. The owner and representation context are fixed for the life of an object so that we can achieve a sound static type system.

On an internal view of an object a we have field access $a.l$, field update $a.l := b$, and method call $a.l(\bar{b})$, where $\bar{b}$ is the tuple of arguments supplied to the method — all fields and methods are accessible using the internal view of an object. When a method is called, the internal view is substituted into the self parameter.

As well as an internal view, we also need an external view of an object. Internal and external views are separated by an protection boundary which hides the representation context analogous to the way existential types hide the type of the representation in the type-theoretic interpretation of abstract data types (for example, see [5], [19]). The syntax of an expression with an existing protection boundary is

$$\textbf{hide}\ \alpha = \pi\ \textbf{in}\ [\ldots]_{\alpha}^{\pi'}$$

The context α, and the value π which it contains, are hidden within $[\ldots]_{\alpha}^{\pi'}$. That is, they can occur within $[\ldots]_{\alpha}^{\pi'}$ but *not* within the expression enclosing the **hide** expression. The expression $\textbf{hide}\ \alpha = \pi\ \textbf{in}\ [\ldots]_{\alpha}^{\pi'}$ is called the external view of the object.

We also write the expression

$$\textbf{new}\ \alpha \subset \pi\ \textbf{in}\ [\ldots]_{\alpha}^{\pi'}$$

to generate a new context for α, immediately inside π, and is used as a new representation context. Each new object instance is built using a **new** expression to initialize its representation context. The above expression evaluates to the following

$$\textbf{hide}\ \alpha = \pi.\kappa\ \textbf{in}\ [\ldots]_{\alpha}^{\pi'}$$

where κ is fresh. Note that π', the owner of this object, need not be the same as π, the bound on the representation, but we require $\pi' \subseteq \pi$.

The key point to note here is that objects have two views: the external view which does not allow access to the representation; and the internal view which can access the entire object.

The **open** expressions subverts the protection boundary of a protected object. It behaves the same way as a similarly named expression for existential types [19]. The syntax is as follows:

$$\textbf{open}\ e\ \textbf{as}\ \alpha, x\ \textbf{in}\ e'$$

e is the expression we wish to open, after it had been evaluated. α is a context variable used inside e' to refer to the hidden context. x is used in e' to refer to the internal view of the opened object.

The **open** expression interacts with the **hide** expression as follows:

$$\textbf{open}\ (\textbf{hide}\ \alpha = \pi\ \textbf{in}\ v)\ \textbf{as}\ \beta, x\ \textbf{in}\ e \quad \leadsto \quad (e\{\!\!\{{}^{v}\!/_{x}\}\!\!\})\{\!\!\{{}^{\pi}\!/_{\beta}\}\!\!\}$$

It binds the representation context π to β and the internal view of the object v to x in the body of expression e, and evaluates the result. A protected object can only be opened within a limited scope, namely that of e on the right hand side of the **in** keyword.

This mechanism allow nearly unlimited subversion of an object, so long as a reference to that object's representation is not stolen. Preventing this theft is the first step toward preventing representation exposure. If subversive activities must happen, then our system makes them explicit. Additional mechanisms on top of our system are required to limit the extent of subversion. One example is the following syntactic restriction.

For maximum protection we specify the following encodings for safe external access to objects (external field access $a_\circ l$, external field update $a_\circ l := b$, and external method call $a_\circ l(\overline{b})$), with the restriction that they are only allowed if they do not allow the representation context, and hence any representation, to escape the scope of the **open** expression.

$$\begin{aligned} a_\circ l &\triangleq \textbf{open } a \textbf{ as } \alpha, x \textbf{ in } x.l \\ a_\circ l := b &\triangleq \textbf{open } a \textbf{ as } \alpha, x \textbf{ in } x.l := b \\ a_\circ l(\overline{b}) &\triangleq \textbf{open } a \textbf{ as } \alpha, x \textbf{ in } x.l(\overline{b}) \end{aligned}$$

The other extension we provide is parameterization over context variables. Analogous to its role in type systems [5], parameterisation gives expressions applicable to any context (or any context satisfying the given bound). These are used for methods which operate regardless of the owner of the objects. In particular, we use parameters for the *new* method for in the encoding of classes [1], so that new objects can be created with any owner. Parameterization is achieved, as usual, by λ-abstractions (with various bounds) over context variables, for example, $\lambda\alpha.e$, $\lambda(\alpha \subseteq p)e$, $\lambda(\alpha \supseteq p)e$. We restrict parameterization to occur only over variables in the owner position.

In the following, we use d^δ to represent a protected object with owner δ, when only the owner is important for discussion.

4 Some Solutions

We demonstrate the expressiveness of the calculus, firstly giving our encoding of containers [6] and then solutions to other problems related to representation exposure.

4.1 Containers

Problem *Adding and removing data from a container*

Motivation Containers are an essential part of object oriented programming. Many abstract data types (sets, sequences, tables), as well as real world examples, such as cars containing a driver and passengers, are containers. Almeida [2] and Hogg [13] consider the data in a container as part of the container's representation. This is *full alias encapsulation* [20]. To add and remove data from a container requires a destructive read, which in our terminology is a change of ownership, and thus requires a change of type, something difficult to handle in a type system. We prefer a different solution.

Solution Consider the data in a container not as part of the representation, but as being owned externally. The protection boundary of the container prevents access to its representation, but allows the externally owned data to pass through.

Example Consider the following car object:

$$car \triangleq \textbf{hide } \alpha = \pi \textbf{ in } [driver = d^\delta, engine = e^\alpha]^{\pi'}_\alpha$$

In this expression π is the actual name of the representation context. The α is used to abstract its real value. The **hide** construct delimits the scope of α, and ultimately π, to the expression on the right hand side of the **in**. The α also appears as a subscript on the object denoting that it is the representation context. The field *engine* has an object whose owner is α which means that this field cannot be accessed or updated from outside this object.

On the other hand, the field *driver* is owned by δ, which must be outside of π for the object to be valid. Thus, *driver* is accessible outside of the object. The *driver* field contains the data of this container. *Engine* contains the representation. The superscript π' is the owner of the *car* object.

The *driver* is accessed with $car_\circ driver$, and updated with an element having the same owner δ using $car_\circ driver := f^\delta$.

The same expressions cannot be used for accessing the *engine*. In particular, an attempt to update the engine using $car_\circ engine := g^?$ fails because we would not know the hidden representation context, so can not give an owner to the expression g.

Classes can be encoded in a manner similar to that used in Abadi and Cardelli's book [1]. By using owner polymorphism as described in Section 4.3, we can create classes parameterized over their owner. This means we can create objects from the class by specifying the owner as a parameter to the constructor.

Flexible alias protection [20] first introduced the idea of separating the data of a container from its representation. This was formalized as ownership types [6].

Limitations All objects stored within a given field must have the same owner.

Mutability of the data may affect things like hashtable keys. Flexible Alias Protection [20] offers some assistance against dependence on an external object's mutable state.

Motto *The protection boundary around an object protects only the representation. Data in a container are not part of the representation.*

4.2 Complex Aggregates

Problem *Accessing representation common to multiple objects within an aggregate.*

Motivation Strict representation containment [6], allowing only external access to the representation through the interface of the representation's owner, limits the kind of abstractions which can be built, precluding even some rudimentary ones.

Example An iterator for a container such as a list where the links of the list are representation, where the iterator, a separate object, needs access to the links. Java's `Enumeration` style iterator is an example. A proxy to the representation, usually adapting the interface.

Solution Use nested objects (or classes) where (1) the owner of the inner object is the same as the owner of the outer object, and (2) the inner object's representation is *inside* the outer object's representation. Condition (1) means that the nested object is accessible in the same places as the original object. Condition (2) implies that the inner object can access the outer object's representation.

We adapt the *car* example to include a *tune-up* object which provides a separate interface, a proxy, to the engine, without exposing the engine to the outside world, and without cluttering the *car* interface with details of the *tune-up* interface.

$$tunacar \triangleq \mathbf{hide}\ \alpha = \pi\ \mathbf{in}\ [driver = d^\delta, engine = e^\alpha,$$
$$tune\text{-}up = \varsigma(x)(\mathbf{new}\ \beta \subset \alpha\ \mathbf{in}\ [\ldots x.engine \ldots]^{\pi'}_{\beta})\ ]^{\pi'}_{\alpha}$$

The object *tunacar* extends the *car* object of the previous example by adding another method *tune-up*. This method creates a new object which has access to the representation of *tunacar* object, because the inner representation context (β) is inside the owner of the outer representation (α).

A call to the method *tune-up* evaluates as follows:

$$\begin{aligned} tunacar_\circ tune\text{-}up &\leadsto \mathbf{new}\ \beta \subset \alpha\ \mathbf{in}\ [\ldots \iota.engine \ldots]^{\pi'}_{\beta} \\ &\leadsto \mathbf{hide}\ \beta = \pi.\kappa\ \mathbf{in}\ [\ldots \iota.engine \ldots]^{\pi'}_{\beta} \end{aligned}$$

where $\iota = [driver = d^\delta, engine = e^\pi, tune\text{-}up = \varsigma(x)\mathbf{new}\ \beta \subset \pi\ \mathbf{in}\ [\ldots x.engine \ldots]^{\pi'}_{\beta}\ ]^{\pi'}_{\pi}$ is the address of the outer object in the object store. The resulting protected object has access to *tunacar*'s engine.

The new object resulting from accessing the *tune-up* field passed through the protection boundary because of a subtle trick with subtyping by which we can express the type independently of α. The reader is urged to consult the first author's thesis, when it becomes available, for complete details.[2] With this object, indirect access to the representation is possible. The absence of the representation context α in its type prevents this representation from being directly exposed.

Each access to *tune-up* generates a new object, thus allowing Java-style iterators which allow different iterators to exist concurrently.

Limitations The proxy or iterator object must be created within the object it whose representation it accessed. Further research is required to see how this feature interacts with inheritance.

Technically, this technique can allow a *delegation*-style interface to the representation, which seemingly defeats the representation containment. Two rebuffs to this point are that (1) the alternate interface must be created on the inside, so it is less likely to occur accidentally; and (2) direct access to the object is not given, so the alternate interface must act as a proxy — this provides a "hook" through which the degree of accessibility can be controlled. Using proxies is a standard solution, our system guarantees that they cannot behave badly.

Motto *Complex aggregates allow multiple objects with access to the same representation without exposing it directly.*

4.3 Owner polymorphism

Problem *Writing methods which are owner-indifferent.*

Motivation We would like methods which are indifferent to the owner of its arguments, performing its calculations without aliasing the arguments to which it is indifferent. Such methods are often static.

[2] The zealous reader may consult the first author directly in the interim.

Example A method for printing objects; a binary method to compare two matrices which is applicable regardless of their owners; the constructor for a class; a *fill* function for a service station operating independently of the vehicle's owner.

Solution Abstracting without bound over the context parameters used in a method gives a method applicable to objects with any owner.

Consider the following method from a the *Service Station* object

$$\mathit{fill} = \varsigma(s)\lambda(\alpha)\lambda(x : Car^{\alpha})e$$

where $x : Car^{\alpha}$ abbreviates the fact that x is a car owned by α. This function is applicable to cars with any owner because the λ abstraction variable α is unbounded. Thus the service station can service cars with different owners using the same method.

When using this function, first supply it with the appropriate context. Then, apply the resulting function to the argument. This is the same way we use polymorphic functions in an explicitly polymorphic calculus (such as System F [9] or Abadi and Cardelli's second-order object calculus [1]).

$$ss_{\circ}\mathit{fill}(\pi)(car^{\pi}) \quad \rightsquigarrow \quad (e\{\!\{^{\pi}/_{\alpha}\}\!\})\{\!\{^{car^{\pi}}/_{x}\}\!\}$$

The most important application of context polymorphic methods is in class constructors. In the object calculus [1], a constructor is a method of the object representing a class. If possible, the *new* method for classes should be context polymorphic so that objects can be created with a given owner and with its data having specified owners. (This is not always possible, for example, if an object's self parameter is assigned to a global variable.)

In the example given here, the parameter α is unbounded. By using a bound in the λ-abstraction, including bounds based on previous context parameters, we can enforce certain structural relationships between the contexts of the arguments passed to a method.

Owner parameterization is related to *roles* from Flexible Alias Protection [20].

Motto *Context parameterization is a step from objects to classses.*

4.4 Initialization

Problem *Initializing an object with externally created parameters which become part of the representation, leaving no external aliases to these parameters.*

Motivation Class constructors often take arguments which become part of the newly created object's representation. Any external reference the these arguments breaks representation containment.

Example A lexer is constructed with an externally created reader so that different implementations of the reader are possible. No external reference to the reader is allowed to remain [7].

Solution There are a number of ways of doing this. Each involves delaying the construction of the argument object until after the creation of the new object. The argument object creation is wrapped as a method in an object, where the method takes a context as a parameter and creates the new object in that context. This wrapper object is passed into the newly created object which calls the method on the representation context, thus creating the argument object to be owned by the representation.

In this example, we use a separate method (*setreader*) for initializing the *lexer*'s *reader* field.

$$\begin{array}{rcl} lexer & = & \mathbf{new}\ \alpha \subset \pi\ \mathbf{in}\ [\ldots, setreader = \varsigma(x)\lambda y(x.reader := y_{\circ}new(\alpha)), \ldots]^{\pi}_{\alpha} \\ reader\_creator & = & \mathbf{new}\ \beta \subset \epsilon\ \mathbf{in}\ [new = \lambda(\alpha)(\mathbf{new}\ \delta \subset \alpha\ \mathbf{in}\ [\ldots (*\ reader\ *) \ldots]^{\alpha}_{\delta})\ ]^{\epsilon}_{\beta} \end{array}$$

The *reader_creator* object has a method *new* which takes a context and creates a reader in that context. This is the delayed construction of the parameter to the lexer.

The body of the *setreader* method passes the representation context to the *new* method of the argument.

To achieve the desired affect, we pass the *reader_creator* object to the *setreader* method of the *lexer* as such:

$$lexer_{\circ}setreader(reader\_creator)$$

The actual reader implementation lies in the body of the λ-abstraction of the *new* method. The remainder is merely packaging to achieve the effect we desire without using first-class functions.

Since the scope of the new representation context (α in *lexer*) is limited — the **new** evaluates to a **hide** expression — the expression prevents external aliasing to the newly created object, which is a problem mentioned in Section 7 of Detlefs et. al. [7].

Limitations The parameter object is created *after* the new object is. This may not always be desirable. The encoding is somewhat convoluted, and would be simplified if first class functions were allowed.

Motto *There is no motto!*

4.5 Subverting the Protection Boundary

Problem *Providing direct but safe access to an object's representation.*

Motivation To implement *friendly functions*, external access to an object's representation is required, but this access must not allow representation to escape. By default, confusion between the representation of different objects must also be avoided.

The second form of protection is not available in languages which support friendly functions of some form (Java, Eiffel, C++, for example), because there is no way distinction between the types of the representation from different objects.

For efficiency reasons, we may also require direct access to some object's representation, possibly with a restricted interface.

Example Operations on matrices, such as addition and equality, require access to the internal arrays underlying the matrix object to perform the operation efficiently. These functions could be implemented as friendly functions and thus require access to the representation of two arguments.

For the lexer example, we require access to the read-only interface of the reader where the *close* method is not available [7].

Solution The solution has two parts: accessing the representation; and providing the restricted interface (though this part is not particular to our system).

The first is simple. Recall the definition of external field access from Section 3, for example. The only expression allowed within the **open** construct was internal field access. By relaxing this restriction, we can apply any expression to the internal view of the object, so long as it does not expose representation, either directly, or by passing it out hidden within another object.

For example, to apply expression e to the representation stored in field rep of object a, use the following:

$$\textbf{open } a \textbf{ as } \alpha, x \textbf{ in } (e\ (x.rep))$$

This expression removes the protection boundary from a, binds the value of its internal view to x and its representation context to α. These are then available to the expression on the right-hand side of the **in** expression. Thus the protected field rep is available in the function application $(e\ (x.rep))$.

The type rules prevent this expression from returning representation. Because the representation context is unknown in the enclosing expression, referred to by α in the expression e, it cannot be captured nor passed into another object.

The implementation of friendly functions employs the above technique at the beginning of the appropriate function body, applying **open** to each argument. The body of the function evaluates in a context where each argument's representation is visible, but distinct.

For example:

$$\lambda a.\lambda b.(\textbf{open } a \textbf{ as } \alpha, x \textbf{ in } (\textbf{open } b \textbf{ as } \beta, y \textbf{ in } e))$$

where x, y, α, and β may appear free in the method body e.

The representations of argument a and argument b cannot be confused because they have different variables, α and β, respectively, in their types as seen in expression e.

For the second part of this problem, we store the representation in either a private or protected field, and use a method to return that representation with a restricted interface. The appropriate encodings are derived from those in Abadi and Cardelli's book. The use of the representation returned by these methods will be limited to the scope of an **open** statement.

Limitations A method which swaps the engines of two cars cannot be implemented if the engines are part of each car's representation. This would be possible is we had change of ownership as discussed in the Future Work section.

Once access to the representation has been gained, the representation can be threaded through function calls, effectively circumventing the protection by creating a dynamic (via stack) alias to the representation.

This technique suffers from syntactic baggage. Perhaps with an alternative syntax we can specify which fields can be subverted in this manner, and then use the dot notation [4] to access only these fields.

Motto *Direct representation access is safely available, by some definition of* safely.

5 Future Work

There are a number of problems we leave for future work ranked in order of increasing difficulty. Some of these are difficult research questions.

Syntax Incorporating the calculus (and its type system) into a programming language like Java will firstly require syntactic modifications similar to those in Pizza [21]. Alternatively, annotations could appear in comments and be processed using a tool such as CoffeeStrainer [3].

Static Inference Static inference based on minimal programmer annotations will make the system more usable. Simple annotations such as `rep` on the fields which contain representation and `subvert` on method arguments (and others) for which we subvert the protection should be sufficient. Then static analysis techniques are then required to determine whether the intended semantics of the annotations are preserved.

Full object calculus Extending to the full object calculus will require either restricted usage of first-class functions and method update, or a looser containment property. An obvious approach is to use only the first-class functions obtained from encodings in the current calculus, similar to function encoding in Abadi and Cardelli's object calculus, and internal method updates. Experience will determine which trade-offs are appropriate.

Effects and Argument Independence Annotations denoting effects (read, write, etc) and argument independence (the parts of an object's interface which are independent of mutable state [20]) provide stronger protection in the presence of aliasing. Such annotations constitute an orthogonal extension to the type system.

It will be instructive to see how easily Greenhouse and Boyland's Object-oriented effects system [11] can be combined with this system.

Efficient dynamic type tests All statically typed object-oriented programming languages have type casting or type tests of some form. This requires type information to be maintained at run-time. To do this correctly for the system presented here requires ownership information to be carried at run-time. In the very least this requires one pointer for each object to its owner. Can this be handled efficiently without losing the desired protection?

Uniqueness Some models guarantee unique object references. A pertinent question is whether to count internal references. Allowing them is congruous with complex aggregates, but it confuses the underlying model and it does not conform with Almeida's work. Other issues worth considering include the difference between stack and store references to objects, and their effect on uniqueness.

Models containing uniqueness include Almeida's Balloons [2], Minsky's u-pointers [18], Hogg's Islands [13], and Boyland's Object-oriented effects system [11].

Uniqueness offers a better solution to the initialization problem.

Change of ownership Change of ownership is important to provide representation containment for systems where objects move from one representation to another, such as in token passing algorithms. In the current system, however, this amounts to changing type, which, in the presence of imperative features, is generally unsound. It seems that the solution would require no more power than providing support for uniqueness.

6 Conclusion

To overcome problems of representation exposure, we extended Abadi and Cardelli's object calculus with object ownership. The calculus has separated the notion of ownership from that of representation containment, giving it greater expressiveness than our earlier approach to object ownership. We have indicated how the calculus directly models both hiding and exposure, or opening, of representation — this allows representation to be exposed, but the exposure must be made explicit and any reference to it can not permanently be stolen. Essentially the calculus model containment with different levels of accessibility. We used the calculus to provide a number of solutions to problems of representation exposure.

Many issues remain unresolved, including the incorporation of the ideas into programming languages in a manner acceptable to programmers. Details of the formal underpinnings of the ideas presented in this paper, including an accompanying type system, are in preparation.

Acknowledgements

The members of the Object Technology Group were invaluable with their comments and support. The referees provided helpful suggestions including the term *owner indifferent.*

References

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects.* Springer-Verlag, 1996.

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

[3] Boris Bokowski. CoffeeStrainer — statically checking structural constraints in Java. Technical Report B-98-14, FU Berlin, Inst. f. Informatik, 1998.

[4] Luca Cardelli and Xavier Leroy. Abstract types and the dot notation. Technical Report SRC-RR-90-56, Digital Systems Research Center, March 1990.

[5] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.

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

[7] David L. Detlefs, K. Rustan M. Leino, and Greg Nelson. Wrestling with rep exposure. Technical Report SRC-RR-98-156, Compaq Systems Research Center, July 1998.

[8] Margaret Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[9] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types.* Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1989.

[10] James Gosling and Frank Yellin. *The Java Application Programming Interface, Volume 1.* Addison-Wesley, 1996.

[11] Aaron Greenhouse and John Boyland. An object-oriented effects system. In *ECOOP'99*, 1999.

[12] C. A. R. Hoare and He Jifeng. A trace model for pointers and objects. Unpublished note.

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

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

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

[16] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development.* The MIT Press, 1986.

[17] Bertrand Meyer. *Eiffel: The Language.* Prentice Hall, 1992.

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

[19] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, 1988.

[20] James Noble, Jan Vitek, and John Potter. Flexible alias protection. In Eric Jul, editor, *ECOOP'98— Object-Oriented Programming*, volume 1445 of *Lecture Notes In Computer Science*, pages 158–185, Berlin, Heidelberg, New York, July 1988. Springer-Verlag.

[21] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, January 1997.

[22] John Potter, James Noble, and David Clarke. The ins and outs of objects. In *Australian Software Engineering Conference*, Adelaide, Australia, November 1998. IEEE Press.