An Object Calculus with Ownership and Containment
Extended Abstract

David Clarke

School of Computer Science and Engineering,
University of New South Wales, Sydney, 2052, Australia
clad@cse.unsw.edu.au*

Abstract

Object ownership and the nesting between objects are arguably important but under appreciated aspects of object-oriented programming. They are also prominent in many alias management schemes. We model object ownership in an extension to Abadi and Cardelli's object calculus [1]. Object owners, called *contexts*, can be created during evaluation allowing ownership on a per object basis. Contexts are partial-ordered to capture the containment relationship between objects.

1 Alias Protection

Aliasing is endemic in object-oriented programming.
NOBLE, VITEK, POTTER [25]

Although fundamental to good object-oriented design [15], aliasing is problematic: when ill-used it breaks the encapsulation necessary for building reliable software components [22]. Therefore, aliasing cannot be eliminated, only managed [20]. A variety of approaches have been proposed to this end. Here we concentrate only on those which limit or report the extent of aliasing by encapsulating all references to certain objects within certain universes (packages, classes, objects, etc.) [3, 4, 9, 11, 18, 19, 21, 25, 24]. This kind of scheme is said to perform *alias encapsulation* [25].

Existing languages such as Java [17] and C++ [12] do not include privacy information as a part of an object's type. This information can be lost and the intended protection can be subverted. The key idea which most alias encapsulation schemes adopt is to *annotate types with protection information which is preserved by subtyping.* For example:

- *Confined types* [4]: modify class types using package-level information to prevent objects from certain classes from being accessed outside a package;
- *Universes* [24]: modify object types based on class-level information to prevent certain objects from being accessed outside of certain classes;
- *Flexible Alias Protection* [25], *Ownership Types* [9], *Universes* [24]: modifies object types based on object-level information to prevent certain objects from being accessed outside certain other objects.

*From March 2001, clad@cs.uu.nl, Department of Computer Science, Utrecht University, Utrecht, The Netherlands.

These schemes select some collection of universes which are used to partition the objects in a system. The universes may be based on packages, classes, or even objects; they are generally nested; and their nesting can be either derived from the scoping of packages or classes,[1] or generated as objects are created. Types are modified to include information about which universe elements of the type inhabits. This is the basis of *ownership*. The universes are then used to control which references are allowed based on their nesting. This is basis of *containment*. Here, we extend Abadi and Cardelli's object calculus to model these notions.

The paper is organised as follows. Section 1 discusses how the representation of an object is protected using a containment invariant based on object ownership. This model was the basis for our earlier work, but it is too inflexible in practice. To overcome this, Section 3 separates the notion of ownership from objects by introducing contexts as the units of ownership and then recasts the containment invariant. The syntax of the calculus is presented in Section 4, followed by some motivating examples in Section 5. We then discuss how the containment invariant is enforced by the type system in Section 6, before presenting the type rules in Section 7, and the dynamic semantics of the calculus in Section 8. Section 9 briefly states its key properties. Section 10 shows some undesirable behaviour the calculus exhibits from a modelling perspective. Section 11 discusses some related work, before the paper concludes in Section 12.

2 Representation, Containment and Ownership

The *representation* of an object is the collection of objects which constitute its internal implementation. The notion that representation should be protected from external access is called *representation containment*, or just *containment.* Assuming that an object system has a root and that access paths from object to object are determined by the fields' contents, we can state a particularly strong formulation of containment, namely, that *all access paths from the root of a system to an object's representation must include that object.* An object is considered to be the *owner* of its representation. The allows the containment property to be restated equivalently as *owners are dominators for access paths between the root of the system and the corresponding representation* [9].

A consequence of this definition is that objects act as single entry points for access to their representation. This

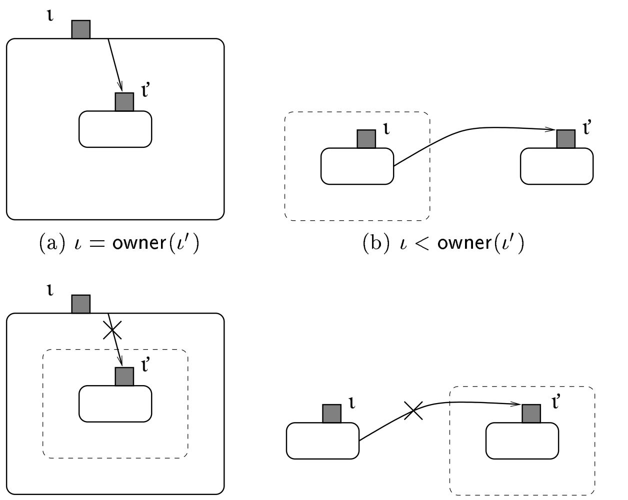
[1] Neither Confined Types [4] nor Universes [24] fully exploit the possibilities.

allows stronger guarantees about an object's invariants, because the only external means for changing its representation is through the object itself; direct external access to the representation is impossible.

Because both formulations above are defined globally over all access paths, neither can be directly incorporated into a type system. We seek a validity condition defined locally between the source and target of a reference, which, when invariant over the entire object graph, induces the containment property.

The first step is to realise that objects when considered as owners form a tree. The root object of the system is the root of the tree, with every object is inside its owner. That is, for all objects ι, we have $\iota \leq \mathsf{owner}(\iota)$ and $\iota \leq \epsilon$. This tree, called the *ownership tree*, captures the nesting between objects.[2]

Now consider the four possibilities depicted in Figure 1. Here a dark square represents an object and the rounded box attached to it encapsulates the object's representation.[3] A dotted box represents an arbitrary number of nested rounded boxes, indicating that the enclosed object is part of the representation of some unspecified other object.



(a) $\iota = \mathsf{owner}(\iota')$ (b) $\iota < \mathsf{owner}(\iota')$

(c) $\iota > \mathsf{owner}(\iota')$ (d) ι and $\mathsf{owner}(\iota')$ are incomparable

Figure 1: Varieties of reference from object ι to object ι'

We wish to exclude references which cross an encapsulation boundary from the outside to the inside, thereby gaining direct external access to an object's representation. The cases excluded are (c) and (d). The remainder, (a) and (b), together give the following containment invariant:

$$\iota \to \iota' \Rightarrow \iota \leq \mathsf{owner}(\iota').$$

where $\to$ denotes the *refers to* relation. When a reference does not satisfy this condition, the object $\mathsf{owner}(\iota')$ can no longer be a dominator for ι' [30], thus violating the intended containment property.

This invariant was uncovered by Potter, Noble and the author [30], and used in a less elegant form in the first system of ownership types [9], although historically the type system came first. This type system was for a class-based language which had neither subclassing or subtyping. Underlying this type system was the model of containment just described. This model is, however, too limiting. We now explore a generalisation.

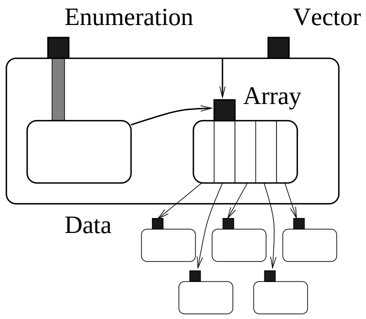


Figure 2: Java's `Vector` with multiple interfaces

3 Contexts and the Containment Invariant

Many object-oriented design idioms cannot be implemented in this model, while retaining some form of representation containment. The reason quite simply is that only one object can both access the representation and be accessed by objects external to the representation.

A common example is iterators in the style of Java's `Enumeration` interfaces [17]. An `Enumeration` object for a `Vector`, for example, has access to the representation of the `Vector` (the underlying array), but is also accessible externally. Thus there are two access paths to the representation, one through the `Vector`, the other through the `Enumerator`, breaking the containment invariant. Figure 2 demonstrates this (as well as a number of other features of our model which we will return to in a moment.)

The set of objects owned by a particular object reside in the same universe (the rounded boxes in the figure). These are the units of ownership which we call *contexts*. Rather than being explicitly tied to objects, contexts form a separate tree to capture the nesting of objects. This is given by a relation $\prec:$, called *inside*, which has maximal element ϵ. To each object we assign two contexts. The owner, $\mathsf{owner}(\iota)$, corresponds to the context in which the object resides. The other context, $\mathsf{rep}(\iota)$, which we call the *representation context*, can be thought of as the owner of the objects ι's representation, or alternatively, where the representation resides. Both contexts are fixed for an object's lifetime to obtain type soundness in the presence of imperative objects.

In the previous model, an object ι owned ι' when $\mathsf{owner}(\iota') = \iota$. Now an object can be thought of as owning objects whose owner is its representation context, that is, when $\mathsf{rep}(\iota) = \mathsf{owner}(\iota')$. By making the corresponding change to the containment invariant above, we obtain our new containment invariant:

$$\iota \to \iota' \ \Rightarrow\ \mathsf{rep}(\iota) \prec: \mathsf{owner}(\iota')$$

This is the invariant which our type system enforces.

The owner context governs which objects can access an object, controlling the target of references, acting like an access control list. Dually, the representation context governs

[2] The ownership tree in fact corresponds to the dominator tree [2].

[3] References originate from the rounded boxes because they come from the implementation of an object. This will become clearer in a moment.

which objects an object can access, controlling the source of references, acting like a capability list.

We require that $\mathsf{rep}(\iota) \prec: \mathsf{owner}(\iota)$ for all objects ι, so that an object can access itself, following the initial model. Indeed, the initial model can be embedded into this system by (1) guaranteeing that each object has a unique representation context, and (2) by ensuring that $\mathsf{rep}(\iota)$ is directly inside $\mathsf{owner}(\iota)$ for all objects ι (for details see [10]). So it is when the representation context is not directly inside the owner context, or when the representation context is not unique, that we obtain flexibility to, for example, implement iterators which can access representation.

In fact, this is exactly what is happening with the `Enumeration` object in Figure 2. Its owner is the same as the owner of `Vector`, but the representation context of the `Enumeration` is inside that of the `Vector`. The grey pipe links the `Vector` to its representation context — it can be though of as extending access to the object to an outer level.

To capture that the representation context controls the source of references and the owner the target, references are drawn from the representation part (round box) to an object, which resides in the owner context. Again references must not cross a boundary from outside of an object to the inside.

The other noteworthy point is that the `Data` objects of the are not considered as part of the `Vector`'s representation. Instead, they are owned externally to the `Vector`. Some other alias management schemes include the `Data` objects as a part of the representation and require destructive read [19] or copy assignment [3] operations to move the `Data` into and out of the `Vector`. Such operations, in our opinion, are unintuitive and problematic.

$$
\begin{array}{rcllll}
\alpha & \in & \textsc{CtxVar} & & & \\
p & \in & \textsc{Context} & ::= & \alpha \mid \pi & \pi \in \mathcal{C} \\
 & & & & & \\
K & \in & \textsc{Perm} & ::= & \langle p \rangle \mid \langle p\!\uparrow \rangle \mid \bigcup[K_1..K_n] & \\
 & & & & & \\
X & \in & \textsc{TypeVar} & & & \\
A, B & \in & \textsc{Type} & ::= & X \mid \mathsf{Top}^K \mid [l_i : \Theta_i{}^{i \in 1..n}]^p_q & \\
 & & & \mid & \mu(X)A \mid \exists(\alpha \prec: p)A & \\
 & & & & & \\
\Theta & \in & \textsc{MtdType} & ::= & A \mid A \rightarrow \Theta \mid \forall(X {<:} A)\Theta & \\
 & & & \mid & \forall(\alpha \prec: p)\Theta \mid \forall(\alpha :\succ p)\Theta & \\
 & & & & & \\
x & \in & \textsc{Var} & & & \\
u, v & \in & \textsc{Value} & ::= & x \mid \iota \mid \mathbf{fold}(A, v) & \\
 & & & \mid & \mathbf{hide}\ p\ \mathbf{as}\ \alpha \prec: q\ \mathbf{in}\ v{:}A & \\
 & & & & & \\
a, b & \in & \textsc{Term} & ::= & v \mid o \mid v.l\langle \Delta \rangle & \\
 & & & \mid & v.l \Leftarrow \varsigma(s : A, \Gamma)b & \\
 & & & \mid & \mathbf{let}\ x : A = a\ \mathbf{in}\ b & \\
 & & & \mid & \mathbf{unfold}(v) & \\
 & & & \mid & \mathbf{expose}\ v\ \mathbf{as}\ \alpha \prec: p, x{:}A\ \mathbf{in}\ b{:}B & \\
 & & & \mid & \mathbf{new}\ \alpha \lhd p\ \mathbf{in}\ a & \\
\Delta & \in & \textsc{Actuals} & ::= & \emptyset \mid v, \Delta \mid A, \Delta \mid p, \Delta & \\
 & & & & & \\
o & \in & \textsc{Object} & ::= & [l_i = \varsigma(s_i : A_i, \Gamma_i)b_i{}^{i \in 1..n}]^p_q & \\
\Gamma & \in & \textsc{Param} & ::= & \emptyset \mid x : A, \Gamma \mid X {<:} A, \Gamma & \\
 & & & \mid & \alpha \prec: p, \Gamma \mid \alpha :\succ p, \Gamma & \\
 & & & & & \\
\sigma & \in & \textsc{Store} & ::= & \emptyset \mid \sigma, \iota \mapsto o & \\
t & \in & \textsc{Config} & ::= & (\Pi, \sigma, a) & \\
\Pi & \in & \textsc{Nesting} & ::= & \emptyset \mid \alpha \prec: p, \Pi &
\end{array}
$$

Figure 3: The Syntax

4 A Calculus with Ownership and Containment

We now present our variant of Abadi and Cardelli's object calculus. The examples presented in Section 5 should help with the intuition. We define the syntax of *contexts*, *permissions*, *types* (including *method types*), *values*, *terms*, and *configurations* in Figure 3. These will be described in turn.

Contexts Contexts are either variables, α, or constants from the partial order $(\mathcal{C}, \prec:_{\mathcal{C}})$ which is supplied as a parameter to the type system. $\mathcal{C}$ may have a maximal element, denoted ϵ. The collection $\mathcal{C}$ could represent the package or class name in a program, to allow ownership per-package or -class.

Permissions Permissions represent the collection of contexts which are accessible in an expression. Expression typing depends on a permission: to access an object or location within an expression its owner context must be included in the permission. We refer to the outermost owner of an object in an expression as the *top level owner*. Only this context matters when accessing an object. Access to the methods is governed by a different context.

There are two basic permissions. The *point* permission $\langle p \rangle$ allows access to the single context p. The *upset* permission $\langle p\!\uparrow \rangle$ allows access to any context q such that $p \prec: q$. Finite unions of permissions can be formed using $\bigcup[K_1..K_n]$. We use the following abbreviations: $\mathsf{void} \ \widehat{=}\ \bigcup[\,]$ and $K \cup K' \ \widehat{=}\ \bigcup[K, K']$.

Permissions also restrict the formation of types and the subtype relation.

Types The types and their intended meanings differ only a little from the standard constructs which they resemble. Types include type variables for both recursive types and type parameterised methods. The top type, Top^K, is the largest type that can be constructed using permission K. It represents the union of all values accessible in an expression given permission K. The object type $[l_i : \Theta_i{}^{i \in 1..n}]^p_q$ lists the names and types of the methods of an object, as well as the owner context p and the representation context q. The precise role p and q play will be discussed is Section 6. Method types, Θ, are described in the next section. Recursive types, $\mu(X)A$, are standard iso-recursive types. Finally, $\exists(\alpha \prec: p)A$ are a limited form of existentially quantified type which abstracts only contexts, but not types.

Method Types Since the objects accessible by an object are different from those accessible outside the object, we can consider objects to have both an inside and outside. This is similar to languages such as Java, where method evaluation can be thought of as occurring inside the target object, since the method can access private members. The object calculus, however, fails to make this distinction — methods with arguments are implemented as functions which can either be invoked or passed to another expression. To remedy this we introduce *fat methods* which must take all of their arguments (values, contexts, types) when the method is called. Thus method types are distinct syntactic entities.

The return type of a method must be a type A. A method can have any number of parameters which can be values, bounded type variables, and context parameters bounded above, and below, giving types $A \to \Theta$, $\forall(X \prec: A)\Theta$, $\forall(\alpha \prec: p)\Theta$ and $\forall(\alpha :\succ p)\Theta$, respectively. The types and contexts in later arguments can depend on those specified in earlier arguments. This implies that, for example, $\forall(\alpha \prec: p)\forall(\beta \prec: \alpha)(A \to B)$ is a valid method type.

Terms Terms are divided into values and expressions. Closed values, that is, those without free term or type variables, constitute the results of evaluation. Expressions are presented in a variant of the *named form* [31]. This simplifies the dynamic semantics and proof of the calculus' properties.

The language is calculus is imperative — aliasing is not problematic in a functional language. Locations, ι, refer to objects in the store.

Objects are given by $[l_i = \varsigma(s_i : A_i, \Gamma_i)b_i{}^{i \in 1..n}]^p_q$, which are object calculus objects extended so that methods take arguments of any kind. The variables s_i are the self parameters used to refer to the current instance of the object in method bodies, b_i. The formal parameters to the method, Γ_i, are a collection of term variables with their type, context variables with their bound, and type variables with their bound. As usual, a method can be considered a field if $s \notin FV(b)$, b is a value, and $\Gamma = \emptyset$. Thus, we do not distinguish between fields and methods. Finally, the owner context is the superscript p, and the subscript q is the representation context.

Method selection, $v.l\langle\Delta\rangle$, denotes a call to method l of object v, with actual parameters, Δ, which are a sequence of values, contexts, and types. All the method arguments must be supplied.

Method update, $v.l \Leftarrow \varsigma(s : A, \Gamma)b$, replaces the method labelled l in the object v with that defined by b.

Folds and **unfold**s are coercions which mediate between the forms of iso-recursive types as usual. Local declarations are defined using **let**.

The hide value, **hide** p **as** $\alpha \prec: q$ **in** $v{:}A$, abstracts context p from the value v and type A, representing it as bound variable α. The context p and, to a certain extent, value v can be thought of as being hidden. The only information known about the hidden context p is that it is inside q. The type of this expression is $\exists(\alpha \prec: q)A$. This term corresponds to the usual term for packing existential types, except that contexts are abstracted rather than types.

The expose expression, **expose** v **as** $\alpha \prec: p, x{:}A$ **in** $b{:}B$, unpacks the contents of a **hide** value. The hidden context and value are substituted for α and x in the expression b. The type system ensures that they satisfy the type and context constraints, and also that the exposed context cannot escape the scope of the expression b. More precisely, it ensures that the hidden context cannot appear in the result type B.

The **hide** and **expose** expressions are more commonly called *pack* and *open* [5], but the names we have chosen are more indicative of their intended behaviour in our modelling: hiding and exposing representation contexts.

The final term, **new** $\alpha \lhd p$ **in** a, creates a new context which is inside but not equal to p, and substitutes it for α in a. Generally a new context will be used to give new objects a unique representation context. **new** is often used in conjunction with a **hide** expression which protects the new context from being exposed.

Contexts appear in both terms and types, but we present them as a single syntactic domain, rather than two as in Flanagan and Abadi [13]. Because contexts are syntactically distinct from other values and types, this presents no confusion.

The contexts **new** generates become a part of a type. This means our type system is a dependent type system. We save ourselves from any problems this could entail by limiting the forms of dependent typing to just contexts, and by disallowing new context variables from appearing free in the type of **new**.

Finally, functions have been omitted to keep the system simple.

Stores and Configurations A configuration, t, represents a snapshot of the evaluation. It consists of an expression, a, a store, σ, and a collection of constraints, Π, on the context variables free in the expression and store. The store maps locations to objects. The constraints, Π, which are all of the form $\alpha \prec: p$, capture the nesting between the contexts created during evaluation.

5 Examples

We now illustrate some of the features of our calculus. To simplify matters, we'll omit annotations and subexpressions which are unnecessary for the discussion.

Simple Objects Consider the following three objects which represent a husband, a wife and their shared car.

$$\begin{array}{rcl} husband &=& [car = car, books = [\cdots]^{bc}]^p_{ours} \\ wife &=& [car = car, CDs = [\cdots]^{cdc}]^p_{ours} \\ car &=& [engine = \cdots]^{ours}. \end{array}$$

The *husband* and *wife* have the same representation context *ours*. This is the owner of the *car*, indicating that the *husband* and the *wife* own the *car*. References to the *car* can only be held by objects with access to the context *ours*, so the level of protection depends on the accessibility of that context.

In addition, the husband is in a book club which has books in context *bc*, and the wife is in a music club which has CDs in context *cdc*. The *husband* can share his *books* with anyone who has access to the book club context *bc*. Similarly, the *wife* can share her *CDs* with anyone who has access to the music club context *cdc*.

Protected Objects and Context Creation The number of objects in a system is not statically determined, in general, so to obtain per object protection we need an operation to dynamically create contexts. For flexibility, this is not tied directly to objects, as the example following will demonstrate.

The expression **new** can create a new context which can be used as the representation context of a new object. This can then be hidden using **hide**, thus creating a representation context accessible only to those within the body of the **new** expression. The new context can also be used to initialise the object's fields. For example

$$\begin{array}{l}\textbf{new}\ \alpha \lhd p\ \textbf{in}\\ \quad \textbf{let}\ engine = [\cdots]^{\alpha}\ \textbf{in}\\ \quad\quad \textbf{let}\ car = [engine = engine, \cdots]^{p}_{\alpha}\ \textbf{in}\\ \quad\quad\quad \textbf{hide}\ \alpha\ \textbf{as}\ \beta \prec: p\ \textbf{in}\ car\end{array}$$

The **new** construct creates a new context just inside p substitutes it for α in the remainder of the expression. An new *engine* object is then created with this context as its owner. Thus the *engine* will become part of the representation of the *car* object, created on the third line. In the last line the actual representation context α is hidden using variable β. The type of this expression will be $\exists(\beta \prec: p)A$, where A is the type of *car*. The hidden context α does not appear free in the type $\exists(\beta \prec: p)A$, following the usual rule for existential type introduction.

The resulting object can be called a protected object because its representation context is hidden. The engine is accessible only using an **expose** expression, but this is limited because no other object can manufacture the necessary context.

It is possible to restrict the syntax of expressions so that every object is created with a unique representation context, as *car* above has, but this is beyond the scope of this paper.

Borrowing Methods with context parameters allow a form of borrowing: otherwise inaccessible objects can be passed to an object and accessed for the duration of a method. Such methods are *owner polymorphic.* Parameters can be bounded above or below.

The following example is a mechanic who will fix a car which can be owned by any context:

$$\begin{array}{l} mycar = [\ldots, fix : B, \ldots]^{myrep} : Car^{myrep}\\ mechanic = [\ldots\ repair = \varsigma(s : A, \alpha \prec: \epsilon, c : Car^{\alpha})c.fix \ldots]\\ mechanic.repair(myrep, mycar)\end{array}$$

Even though the *mechanic* may not generally have access to *mycar*, access is allowed by passing the context *myrep*, the owner of *mycar*, to the *mechanic*'s *repair* method for the duration of that method.

Classes Classes can follow the encodings in [1], with minor modifications to deal with ownership. The following example is indicative of the modifications require.

An unbounded stack class is based on the following types (simplified somewhat to maintain brevity):

$$\begin{array}{rcl} \text{CSTACK} &=& [new : \forall(\alpha \prec: \epsilon)\forall(\beta :\succ \alpha)\forall(X{<:}\mathsf{Top}^{\langle\beta\rangle})\text{STACK}]^{\epsilon}_{\epsilon}\\ \text{STACK} &=& \exists(\gamma \prec: \alpha)[head : \text{LINK}, push : X \to (), pop : X]^{\alpha}_{\gamma}\\ \text{LINK} &=& \mu(Y)[next : Y, data : X]^{\gamma}_{\gamma}\end{array}$$

A CSTACK, the type of the stack class, has one method for creating stacks. This takes arguments for the owner of the stack, α, the owner of the data in the stack, β and the type of the data, X. The constraints on these imply that α can be any context, that β is accessible to the to-be-created representation context, which will be inside α, and that X will be any type visible with just permission $\langle\beta\rangle$, that is any type with top level owner β. The CSTACK class (*classStack* below) is accessible to all objects because its owner is ϵ.

STACK is the type of stacks, with free parameters which will be filled in when CSTACK's *new* method is selected. Its hidden representation context γ is created afresh for each stack object inside α. A better type would make the *head* field private.

LINK, the type of links between elements of the stack, is a recursive type because each *next* field also has type LINK. This means that all links have the same owner, and hence receive the same amount of protection.

Neither CSTACK nor LINK have distinct owner and representation contexts, because they do not have any representation.

Finally, the code for the class of stacks is:

$$\begin{array}{l} classStack =\\ \ [new = \varsigma(z : \text{CSTACK}, \alpha \prec: \epsilon, \beta :\succ \alpha, X{<:}\mathsf{Top}^{\langle\beta\rangle})\\ \ \ \textbf{new}\ \gamma \lhd \alpha\ \textbf{in}\\ \ \ \ \textbf{hide}\ \gamma\ \textbf{as}\ \gamma \prec: \alpha\ \textbf{in}\\ \ \ \ \ [head = \varsigma(s : \text{STACK})s.head,\\ \ \ \ \ \ push = \varsigma(s : \text{STACK}, d : X)\\ \ \ \ \ \ \ s.head := \textbf{fold}(\text{LINK}, [next = s.head, data = d]^{\gamma}_{\gamma}),\\ \ \ \ \ \ pop = \varsigma(s : \text{STACK})\\ \ \ \ \ \ \ \textbf{let}\ prev = \textbf{unfold}(head)\ \textbf{in}\\ \ \ \ \ \ \ \ s.head := prev.next;\ prev.data\ ]^{\alpha}_{\gamma}\quad]^{\epsilon}_{\epsilon}\end{array}$$

where the *head* method uses Abadi and Cardelli's *undefined but well-typed* idiom, and ";" is sequential composition which can be encoded in the standard manner using **let** expressions [1].

Multiple Interfaces Just as a `Vector` presents different interfaces to its clients, we could imagine a car abstraction presenting at least two different interfaces to its users. The principal interface is the driver's, and the other, less frequently used one, is for mechanics to tune the car's engine. Each interface consists of its own components, thus is a separate object with its own representation; each interface accesses the protected engine of the car.

At first glance it would seem that, following the first example in this section, sharing the principal object's representation context would suffice, but this prevents each interface from having its own representation. The key to achieving the desired model is illustrated by the following example:

$$\begin{array}{l} mycar =\\ \ \textbf{new}\ \alpha \lhd p\ \textbf{in}\\ \ \ \textbf{let}\ engine = [\cdots, tamper = \cdots]^{\alpha}\ \textbf{in}\\ \ \ \ \textbf{let}\ car =\\ \ \ \ \ [engine = \varsigma(x)engine,\\ \ \ \ \ \ tune = \varsigma(s)\\ \ \ \ \ \ \ \ \textbf{let}\ t =\ \textbf{new}\ \beta \lhd \alpha\ \textbf{in}\ [\ldots s.engine.tamper \ldots]^{p}_{\beta}\ \textbf{in}\\ \ \ \ \ \ \ \ \ \textbf{hide}\ \beta\ \textbf{as}\ \gamma \prec: p\ \textbf{in}\ t]^{p}_{\alpha}\ \textbf{in}\\ \ \ \ \textbf{hide}\ \alpha\ \textbf{as}\ \delta \prec: p\ \textbf{in}\ car\end{array}$$

The *mycar* object itself is the principal. The alternative interface is a new object created by the *tune* method. The *engine* is part of *mycar*'s representation. The representation context of the new *tune* object is a new context β which is inside the representation context of the *mycar* object. This means that the *tune* object has permission to tamper with the engine.

Recapping, the idea is to create the representation context of the interface objects inside that of the principal, and give both the same owner.

Accessing Representation and Friendly Functions The **expose** expression allows direct, but limited, access to the hidden representation context. This is useful for implementing friendly functions which access the representation of two or more different objects.

Dealing with friendly functions in the presence of representation containment is one of the limitations of most existing alias management schemes; although essential for applications, friendly functions violate encapsulation. (They also present a number of challenges for typing [28, for example].)

In the present setting, the key issue is to avoid confusing the representation of the different objects involved in a friendly function. This was impossible in our previous work [9], because there was no way to distinguish two different representation contexts. The solution Universes adopts restricts access to all but one party to read-only [24], but this is potentially too restrictive.

The solution in the calculus presented here uses nested **expose** statements to access the representation context of each member of the friendship. The following brief example accesses the representation context of two engines to compare the rate of their exhaust emissions:

$$
\begin{array}{l}
[\mathit{analyse} = \varsigma(s : A, \mathit{car1} : \exists(\alpha \prec: p)\,\mathit{Car}, \mathit{car2} : \exists(\beta \prec: p)\,\mathit{Car'}) \\
\quad \textbf{expose}\ \mathit{car1}\ \textbf{as}\ \alpha \prec: p, c1 : \mathit{Car}\ \textbf{in} \\
\quad \textbf{expose}\ \mathit{car2}\ \textbf{as}\ \beta \prec: p, c2 : \mathit{Car'}\ \textbf{in} \\
\qquad c1.\mathit{engine.exhaust} > c2.\mathit{engine.exhaust}]
\end{array}
$$

The representation contexts cannot be confused because they are bound to different variables, α and β, and thus the types of the representation will differ in their owner parameters.

6 From the Containment Invariant to Types

We now demonstrate how the containment invariant is enforced in the type system. Recall that the containment invariant states a necessary condition for a reference from ι to ι' to exist:

$$\iota \rightarrow \iota' \;\Rightarrow\; \mathsf{rep}(\iota) \prec: \mathsf{owner}(\iota').$$

Let $\iota \mapsto [l_i = \varsigma(s_i : A)b_i{}^{i\in 1..n}]^p_q$ be a location-object binding in some store. The locations accessible to ι are those appearing in the method bodies b_i. The containment invariant can be transformed to give an upper bound on these locations:

$$\{\iota' \mid q \prec: \mathsf{owner}(\iota')\}, \tag{1}$$

where $q = \mathsf{rep}(\iota)$. The permission $\langle q\!\uparrow\rangle$ corresponds to the set $\{p \mid q \prec: p\}$. Using this (1) becomes:

$$\{\iota' \mid \mathsf{owner}(\iota') \in \langle q\!\uparrow\rangle\}. \tag{2}$$

Now consider an object at location ι' having owner p'. Access to this object requires permission $\langle p'\rangle$, which corresponds to the singleton set $\{p'\}$. Thus ι' is in the set (2) if and only if

$$\langle p'\rangle \subseteq \langle q\!\uparrow\rangle. \tag{3}$$

This condition is enforced by our type system.

Expressions are typed against a permission which limits the object owners accessible in the expression. A locations and object whose owner is not in the permission cannot be accessed. The following (simplified) object typing rule contains the essential ingredients for attaining the containment invariant:

(Val Object-Simplified) (where $A \equiv [l_i : \Theta_i{}^{i\in 1..n}]^p_q$)

$$\frac{E, s_i : A; \langle q\!\uparrow\rangle \vdash b_i : \Theta_i \quad \forall i \in 1..n}{E; \langle p\rangle \vdash [l_i = \varsigma(s_i : A)b_i{}^{i\in 1..n}]^p_q : A}$$

The conclusion states, among other things, that the permission required to access this object is $\langle p\rangle$, where p is its owner. This captures the left-hand side of condition (3). The premises, $i \in 1..n$, each state that the permission governing access in the method bodies b_i is $\langle q\!\uparrow\rangle$, where q is the representation context. This captures the right-hand side of condition (3). Therefore, the only locations accessible in a method body are those permitted by the containment invariant.

This completes the details of the local formulation we sort. In fact, a global component still remains. The ownership tree is captured by the contexts defined in the environment E. But the containment invariant is defined locally using permissions.

7 Type Rules

The type rules formally capture the intuition described above. Due to space limitations, we describe only the most important features of the type system.

The type system depends on a *typing environment*, E, which records the types of program variables and locations, subtyping assumptions for type variables, and lower or upper bounds on context variables. The typing environment is organised as a sequence of bindings and constraints, where $\emptyset$ denotes the empty environment:

$$
\begin{array}{lcl}
E & ::= & \emptyset \mid E, x : A \mid E, \iota : A \mid E, X {<:} A \\
 & \mid & E, \alpha \prec: p \mid E, \alpha :\succ p
\end{array}
$$

Note that syntax of method formal parameters, Γ, and of the constraints, Π, are included within this syntax. This allows them to be treated uniformly in the type rules.

The type system is based on the twelve judgements given in Figure 4. The well-typed term, well-formed type, and the subtype relation judgements depend on a permission K. This has the following consequences:

- Expressions can access only contexts given in the permission.
- All values of a type which is defined for some permission are accessible in expressions having that permission.
- The subtyping relation ensures that all subtypes of a type are valid with the same permissions as the type.

Figure 5 contains the rules for well-formed environments.

Figure 6 defines well-formed contexts and their nesting. The rules (Context π) and (In π) simply embeds the constant contexts and their nesting into contexts and their nesting. The rule (In ϵ) guarantees that ϵ is maximal. If $\mathcal{C}$ has no maximal element, then this rule is omitted.

Well-formed Permission and Subpermissions Figure 7 defines well-formed permissions and the subpermission relation.

A point permission is a subpermission of the corresponding upset permission by rule (SubPerm p). The rule (SubPerm $\prec:$), in effect, lifts the nesting relation between contexts to upset permissions, based on the following intuition:

$E \vdash \diamond$	*E is a well-formed typing environment*
$E \vdash p$	*p is a well-formed context in E*
$E \vdash p \prec: q$	*p is inside q in E*
$E \vdash K$	*K is a well-formed permission in E*
$E \vdash K \subseteq K'$	*K is a subpermission of K′ in E*
$E; K \vdash A$	*A is a well-formed type in E given permission K*
$E; K \vdash A{<:}B$	*A is a subtype of B in E given permission K*
$E; K \vdash a : A$	*a is a well-typed expression of type A in E given permission K*
$E; K \vdash \Theta$ *meth*	*Θ is a well-formed method type in E given permission K.*
$E; K \vdash (\Theta)(\Delta) \Rightarrow C$	*Arguments Δ match Θ with return type C in E given permission K*
$E \vdash \sigma$	*σ is a well-formed store in E*
$E; K \vdash (\Pi, \sigma, a) : A$	*(Π, σ, a) is a well-typed configuration with expression type A in E given permission K*

Figure 4: Judgements

assume that $E \vdash \langle p\!\uparrow\rangle \subseteq \langle q\!\uparrow\rangle$; if $p' \in \langle p\!\uparrow\rangle$, then $p \prec: p'$; from $q \prec: p$, we have $q \prec: p'$; hence $p' \in \langle q\!\uparrow\rangle$.

Rules (Perm Union), (SubPerm Union-L), and (SubPerm Union-UB) extend permissions and the subpermission relation to unions (following, for example, Pierce [27].)

Whenever $E \vdash K \subseteq K'$ holds, any well-formed term, type, or subtype defined given permission K is also well-defined with the larger permission K'.

Well-formed Types Figure 8 defines well-formed types.

The rule (Type X) helps ensure that type substitution does not violate permissions by being valid for permissions required to construct the bound on a type variable.

The constraints in rule (Type Object) are based on the discussion in Section 6. The additional condition $E \vdash q \prec: p$ implies that $E \vdash \langle p\rangle \subseteq \langle q\!\uparrow\rangle$, ensuring that an object can access itself.

Existential type formation is standard, by (Type Exists), except that it is restricted to contexts with an upper bound. The type rule has an additional clause, $E \vdash K$, which guarantees that the existential type cannot abstract a context in the top level owner position. Doing so would break our desired containment invariant, because access control is based on the top level owner position. We now briefly illustrate how the clause $E \vdash K$ achieves this.

An example of the type we wish to avoid is $\exists(\alpha \prec: p)[\cdots]^{\alpha}_{q}$. We argue that such a type is impossible. Consider the following typing derivation, where $\vdash_?$ indicates questionable judgements, and *1* and *2* denote application of the rules (Type Lift) and (Type Exists), respectively:

$$\cfrac{\cfrac{E, \alpha \prec: p; \langle\alpha\rangle \vdash [\cdots]^{\alpha}_{q} \quad E, \alpha \prec: p \vdash_? \langle\alpha\rangle \subseteq K}{E, \alpha \prec: p; K \vdash_? [\cdots]^{\alpha}_{q}}\,1 \qquad E \vdash K}{E; K \vdash_? \exists(\alpha \prec: p)[\cdots]^{\alpha}_{q}}\,2$$

There is no K such that $E \vdash K$ and $E, \alpha \prec: p \vdash \langle\alpha\rangle \subseteq K$. Note that $\alpha \notin \mathsf{dom}(E)$ and hence $\alpha \notin FV(K)$. If K is a union, then one of its components must satisfy the condition we require. Clearly K cannot be of the form $\langle p'\rangle$, because this would require that $p' = \alpha$, which is disallowed. Hence K must be of the form $\langle p'\!\uparrow\rangle$. If $E, \alpha \prec: p \vdash_? \langle\alpha\rangle \subseteq \langle p'\!\uparrow\rangle$, then $E, \alpha \prec: p \vdash_? \langle\alpha\!\uparrow\rangle \subseteq \langle p'\!\uparrow\rangle$, from which follows $E, \alpha \prec: p \vdash_? p' \prec: \alpha$. But this cannot be derived.

The rule (Type Rec) for recursive types places the given bound of the variable X to ensure that the unfolding of type $\mu(X)A$ into $A\{\!\{^{\mu(X)A}/_{X}\}\!\}$ is well-formed, because the permission K required to type X is sufficient for typing $\mu(X)A$.

By (Type Lift) a type which is valid with some permission is also valid given a larger permission.

Well-formed Method Types Method types include function types, and types parameterised by type and context variables, both with appropriate bounds. The type rules are given in Figure 9. The two rules (Type All $\prec:$) and (Type All $:\succ$) allow the permission to be extended by the point permission corresponding to the context parameter, so that an additional context is accessible for the duration of the method.

Making method types variant does not present any challenges [1], nor does adding first class functions (of each kind). In the latter case, the separation of methods and functions must remain to preserve the distinction between evaluation inside or outside an object. This may result in some redundancy in the definitions.

Well-formed Subtype Relation The subtyping rules in Figure 10 contain no surprises. The rule (Sub Object) allows neither the owner nor the representation context to vary, only which methods are present. The rule (Sub Exists) follows the pattern of subtyping existential types, except that it applies only to contexts, and the $\prec:$ relation lacks a notion of subsumption. Finally, (Sub Lift) states that subtyping relationships valid for some permission are valid at a larger permissions.

Well-typed Terms Figure 11 defines well-typed terms. Expressions are typed against a typing environment and a permission. The permission bounds the contexts appearing in the top-level owner position of the objects and locations in the given expression. Term typing depends on the following auxiliary functions.

The function $\lfloor\Gamma\rfloor_C$ converts the method arguments Γ and the return type C into a method type Θ:

Definition 1 ($\lfloor\Gamma\rfloor_C$)

$$\begin{aligned} \lfloor\emptyset\rfloor_C &\;\widehat{=}\; C \\ \lfloor x : A, \Gamma\rfloor_C &\;\widehat{=}\; A \to \lfloor\Gamma\rfloor_C \\ \lfloor X{<:}A, \Gamma\rfloor_C &\;\widehat{=}\; \forall(X{<:}A)\lfloor\Gamma\rfloor_C \end{aligned}$$

$$\frac{}{\emptyset \vdash \diamond}\ (\text{Env } \emptyset) \qquad \frac{E;K \vdash A \quad X \notin \mathsf{dom}(E)}{E, X{<:}A \vdash \diamond}\ (\text{Env } X) \qquad \frac{E;K \vdash A \quad x \notin \mathsf{dom}(E)}{E, x : A \vdash \diamond}\ (\text{Env } x)$$

$$\frac{E \vdash p \quad \alpha \notin \mathsf{dom}(E)}{E, \alpha \prec: p \vdash \diamond}\ (\text{Env } \alpha \prec:) \qquad \frac{E \vdash p \quad \alpha \notin \mathsf{dom}(E)}{E, \alpha :\succ p \vdash \diamond}\ (\text{Env } \alpha :\succ) \qquad \frac{E; \langle p \rangle \vdash [l_i : {\Theta_i}^{i \in 1..n}]^p_q \quad \iota \notin \mathsf{dom}(E)}{E, \iota : [l_i : {\Theta_i}^{i \in 1..n}]^p_q \vdash \diamond}\ (\text{Env Location})$$

Figure 5: Well-formed Environments

$$\frac{E \vdash \diamond \quad \pi \in \mathcal{C}}{E \vdash \pi}\ (\text{Context } \pi) \qquad \frac{\alpha \in \mathsf{dom}(E)}{E \vdash \alpha}\ (\text{Context } \alpha) \qquad \frac{\alpha \prec: p \in E}{E \vdash \alpha \prec: p}\ (\text{In} \prec:) \qquad \frac{\alpha :\succ p \in E}{E \vdash p \prec: \alpha}\ (\text{In} :\succ) \qquad \frac{E \vdash \diamond \quad \pi \prec:_{\mathcal{C}} \pi'}{E \vdash \pi \prec: \pi'}\ (\text{In } \pi) \qquad \frac{E \vdash p}{E \vdash p \prec: \epsilon}\ (\text{In } \epsilon)$$

Figure 6: Well-formed contexts and their nesting (reflexivity and transitivity of $\prec:$ omitted)

$$\frac{E \vdash p}{E \vdash \langle p \rangle}\ (\text{Perm } p) \qquad \frac{E \vdash p}{E \vdash \langle p\!\uparrow \rangle}\ (\text{Perm } p \uparrow) \qquad \frac{E \vdash K_i \quad \forall i \in 1..n}{E \vdash \bigcup[K_1..K_n]}\ (\text{Perm Union}) \qquad \frac{E \vdash p}{E \vdash \langle p \rangle \subseteq \langle p\!\uparrow \rangle}\ (\text{SubPerm } p) \qquad \frac{E \vdash q \prec: p}{E \vdash \langle p\!\uparrow \rangle \subseteq \langle q\!\uparrow \rangle}\ (\text{SubPerm} \prec:)$$

$$\frac{E \vdash K_i \subseteq K \quad \forall i \in 1..n}{E \vdash \bigcup[K_1..K_n] \subseteq K}\ (\text{SubPerm Union-L}) \qquad \frac{E \vdash \bigcup[K_1..K_n] \quad i \in 1..n}{E \vdash K_i \subseteq \bigcup[K_1..K_n]}\ (\text{SubPerm Union-UB})$$

Figure 7: Permissions and Sub-permission (reflexivity and transitivity of $\subseteq$ omitted)

$$\begin{array}{rcl} \lfloor \alpha \prec: p, \Gamma \rfloor_C & \widehat{=} & \forall(\alpha \prec: p)\lfloor \Gamma \rfloor_C \\ \lfloor \alpha :\succ p, \Gamma \rfloor_C & \widehat{=} & \forall(\alpha :\succ p)\lfloor \Gamma \rfloor_C \end{array}$$

The function $\lceil \Gamma \rceil$ gives the collection of additional permissions for the context parameters in Γ:

Definition 2 ($\lceil \Gamma \rceil$)

$$\begin{array}{rcl} \lceil \emptyset \rceil & \widehat{=} & \mathit{void} \\ \lceil x : A, \Gamma \rceil & \widehat{=} & \lceil \Gamma \rceil \\ \lceil X{<:}A, \Gamma \rceil & \widehat{=} & \lceil \Gamma \rceil \\ \lceil \alpha \prec: p, \Gamma \rceil & \widehat{=} & \langle \alpha \rangle \cup \lceil \Gamma \rceil \\ \lceil \alpha :\succ p, \Gamma \rceil & \widehat{=} & \langle \alpha \rangle \cup \lceil \Gamma \rceil \end{array}$$

In (Val Object), the permission $\langle p \rangle$ is required to create an object with owner context p. The method body b_i of method $\varsigma(s_i : A, \Gamma_i)b_i$ is typed against an environment extended with the self parameter s_i with A, the type of the object being formed, and with the formal parameter list Γ_i, and against the union of permission $\langle q\!\uparrow \rangle$, where q is the representation context, the point permissions for any context declared in Γ_i. Because the contexts in Γ_i are variables, no location with such an owner can appear in the method body b_i. Thus this does not effect the containment invariant. The return type of the method body C_i and the formal parameters are combined to give the method type Θ_i.

In (Val Select), given that the selected method of a well-typed target has type Θ_j, the clause $E; K \vdash (\Theta_j)(\Delta) \Rightarrow C_j$ guarantees that the arguments Δ are correct in number and type, and that they and the return type C_j are all well-formed given permission K. This ensures that enough permission is given to access the method's arguments and return value. The type rules for well-typed arguments lists are given in Figure 12 and are explained in the next section.

In (Val Update) the new method body is typed against a permission K' which, in effect, is the intersection of the contexts accessible inside the object, $\langle q \uparrow \rangle$, and the contexts visible to the surrounding expression, K, along with the point permissions for the context parameters declared in Γ_j. This 'intersection' of permissions prevents any otherwise inaccessible locations from being added into object, while maintaining the constraints on the expression performing the method update.

The first clause in (Val New) types the term a with the additional assumption about the new context α. The permission is extended with $\langle \alpha \rangle$ so that objects can be created with this new context as owner, as in the Protected Objects example from Section 5. The second clause prevents the new context α from appearing in the resulting type.

The rule (Val Hide) parallels the standard rule for packing existential types, with the additional clause $E; K \vdash \exists(\alpha \prec: q)A$ for the reasons discussed above. Similarly, (Val Expose) parallels the usual rule for opening existential types.

$$\frac{X{<:}A \in E \quad E;K \vdash A}{E;K \vdash X}\ \text{(Type } X\text{)} \qquad \frac{E \vdash K}{E;K \vdash \mathsf{Top}^K}\ \text{(Type Top)} \qquad \frac{E;\langle q\!\uparrow\rangle \vdash \Theta_i\ \ meth \quad \forall i \in 1..n \quad E \vdash q \prec: p}{E;\langle p\rangle \vdash [l_i : \Theta_i{}^{i \in 1..n}]_q^p}\ \text{(Type Object) } (l_i \text{ distinct})$$

$$\frac{E, X{<:}\mathsf{Top}^K; K \vdash A}{E;K \vdash \mu(X)A}\ \text{(Type Rec)} \qquad \frac{E, \alpha \prec: p; K \vdash B \quad E \vdash K}{E;K \vdash \exists(\alpha \prec: p)B}\ \text{(Type Exists)} \qquad \frac{E;K \vdash A \quad E \vdash K \subseteq K'}{E;K' \vdash A}\ \text{(Type Lift)}$$

Figure 8: Well-formed Types

$$\frac{E;K \vdash A}{E;K \vdash A\ \ meth}\ \text{(Type Return)} \qquad \frac{E;K \vdash A \quad E;K \vdash \Theta\ \ meth}{E;K \vdash A \to \Theta\ \ meth}\ \text{(Type Arrow)} \qquad \frac{E, X{<:}A; K \vdash \Theta\ \ meth}{E;K \vdash \forall(X{<:}A)\Theta\ \ meth}\ \text{(Type All)}$$

$$\frac{E, \alpha \prec: p; K \cup \langle\alpha\rangle \vdash \Theta\ \ meth \quad E \vdash K}{E;K \vdash \forall(\alpha \prec: p)\Theta\ \ meth}\ \text{(Type All } \prec:\text{)} \qquad \frac{E, \alpha :\succ p; K \cup \langle\alpha\rangle \vdash \Theta\ \ meth \quad E \vdash K}{E;K \vdash \forall(\alpha :\succ p)\Theta\ \ meth}\ \text{(Type All } :\succ\text{)}$$

Figure 9: Well-formed Method Type

$$\frac{X{<:}A \in E \quad E;K \vdash A}{E;K \vdash X{<:}A}\ \text{(Sub } X\text{)} \qquad \frac{E;K \vdash A}{E;K \vdash A{<:}\mathsf{Top}^K}\ \text{(Sub Top)} \qquad \frac{E;\langle q\!\uparrow\rangle \vdash \Theta_i\ \ meth \quad \forall i \in 1..n+m \quad E \vdash q \prec: p}{E;\langle p\rangle \vdash [l_i : \Theta_i{}^{i \in 1..n+m}]_q^p{<:}[l_i : \Theta_i{}^{i \in 1..n}]_q^p}\ \text{(Sub Object) } (l_i \text{ distinct})$$

$$\frac{E;K \vdash \mu(X)A \quad E;K \vdash \mu(Y)B \quad E, Y{<:}\mathsf{Top}^K, X{<:}Y; K \vdash A{<:}B}{E;K \vdash \mu(X)A{<:}\mu(Y)B}\ \text{(Sub Rec)}$$

$$\frac{E \vdash p \prec: p' \quad E, \alpha \prec: p; K \vdash A{<:}A' \quad E \vdash K}{E;K \vdash \exists(\alpha \prec: p)A{<:}\exists(\alpha \prec: p')A'}\ \text{(Sub Exists)} \qquad \frac{E;K \vdash A{<:}B \quad E \vdash K \subseteq K'}{E;K' \vdash A{<:}B}\ \text{(Sub Lift)}$$

Figure 10: Well-formed Subtype Relation (reflexivity and transitivity of <: omitted)

$$\frac{x : A \in E \quad E;K \vdash A}{E;K \vdash x : A}\ \text{(Val } x\text{)}$$

$$\frac{\iota : [l_i : \Theta_i{}^{i\in 1..n}]^p_q \in E}{E;\langle p\rangle \vdash \iota : [l_i : \Theta_i{}^{i\in 1..n}]^p_q}\ \text{(Val Location)}$$

(Val Object) (where $A \equiv [l_i : \Theta_i{}^{i\in 1..n}]^p_q$ and $\Theta_i \equiv \lfloor \Gamma_i \rfloor_{C_i}$)

$$\frac{E, s_i : A, \Gamma_i; \langle q\uparrow\rangle \cup \lceil \Gamma_i \rceil \vdash b_i : C_i \quad \forall i \in 1..n}{E;\langle p\rangle \vdash [l_i = \varsigma(s_i : A, \Gamma_i) b_i{}^{i\in 1..n}]^p_q : A}$$

(Val Select)

$$\frac{E;K \vdash v : [l_i : \Theta_i{}^{i\in 1..n}]^p_q \quad E;K \vdash (\Theta_j)(\Delta) \Rightarrow C_j \quad j \in 1..n}{E;K \vdash v.l_j\langle\Delta\rangle : C_j}$$

(Val Update) (where $A \equiv [l_i : \Theta_i{}^{i\in 1..n}]^p_q$ and $\Theta_j \equiv \lfloor \Gamma_j \rfloor_{C_j}$)

$$\frac{\begin{array}{c} E;K \vdash v : A \quad E, s : A, \Gamma_j; K' \vdash b : C_j \\ E, \Gamma_j \vdash K' \subseteq \langle q\uparrow\rangle \cup \lceil \Gamma_j \rceil \quad E, \Gamma_j \vdash K' \subseteq K \cup \lceil \Gamma_j \rceil \quad j \in 1..n \end{array}}{E;K \vdash v.l_j \Leftarrow \varsigma(s : A, \Gamma_j) b : A}$$

(Val Fold) (where $A \equiv \mu(X)B$)

$$\frac{E;K \vdash v : B\{\!\{^A/_X\}\!\}}{E;K \vdash \mathbf{fold}(A, v) : A}$$

(Val Unfold) (where $A \equiv \mu(X)B$)

$$\frac{E;K \vdash v : A}{E;K \vdash \mathbf{unfold}(v) : B\{\!\{^A/_X\}\!\}}$$

(Val Let)

$$\frac{E;K \vdash a : A \quad E, x : A; K \vdash b : B}{E;K \vdash \mathbf{let}\ x : A = a\ \mathbf{in}\ b : B}$$

(Val New)

$$\frac{E, \alpha \prec: p; K \cup \langle\alpha\rangle \vdash a : A \quad E;K \vdash A}{E;K \vdash \mathbf{new}\ \alpha \lhd p\ \mathbf{in}\ a : A}$$

(Val Hide)

$$\frac{E \vdash p \prec: q \quad E;K \vdash v\{\!\{^p/_\alpha\}\!\} : A\{\!\{^p/_\alpha\}\!\} \quad E;K \vdash \exists(\alpha \prec: q)A}{E;K \vdash \mathbf{hide}\ p\ \mathbf{as}\ \alpha \prec: q\ \mathbf{in}\ v{:}A : \exists(\alpha \prec: q)A}$$

(Val Expose)

$$\frac{E;K \vdash v : \exists(\alpha \prec: p)A \quad E;K \vdash B \quad E, \alpha \prec: p, x : A; K \cup \langle\alpha\rangle \vdash b : B}{E;K \vdash \mathbf{expose}\ v\ \mathbf{as}\ \alpha \prec: p, x{:}A\ \mathbf{in}\ b{:}B : B}$$

(Val Subsumption)

$$\frac{E;K \vdash a : A \quad E;K' \vdash A <: B \quad E \vdash K \subseteq K'}{E;K' \vdash a : B}$$

Figure 11: Well-typed Expressions

Finally, (Val Subsumption) allowing an expression given one type to be given a supertype and to be used with a larger permission.

Well-typed Actual Parameters The type rules in Figure 12 check that the arguments supplied to a method are correct in number and type. Underlying the rules are the usual rules for function and type application. The judgement $E \vdash (\Theta)(\Delta) \Rightarrow C$ actually resembles (total) function application. All types and values, including the return type, must be accessible given permission K.

Well-formed Stores and Configurations The type rules for stores and configurations are given in Figure 13. Configuration typing is performed in an environment consisting of two parts: Π to account for the nesting of free context variables and E' for location types. The remainder is standard.

8 Dynamic Semantics

The operational semantics of the calculus is presenting in a big-step, substitution-based style in Figure 14. Fundamentally it differs little from the object calculus semantics of Gordon et al. [16], though the named form of expression allows some minor simplifications. Again, we only highlight key and subtle points.

The evaluation rule (Subst Select) uses the following definition to construct a substitution from the actual parameters Δ into the formal parameters Γ for the selected method. The substitution into the method body b is then performed and the resulting expression evaluated.

Definition 3 ($\{\!\{\Delta/\Gamma\}\!\}$)

$$\begin{array}{rcl} \{\!\{\emptyset/\emptyset\}\!\} & \widehat{=} & \epsilon \\ \{\!\{v, \Delta/x : A, \Gamma\}\!\} & \widehat{=} & \{\!\{^v/_x\}\!\}\{\!\{\Delta/\Gamma\}\!\} \\ \{\!\{B, \Delta/X <: A, \Gamma\}\!\} & \widehat{=} & \{\!\{^B/_X\}\!\}\{\!\{\Delta/\Gamma\}\!\} \\ \{\!\{p, \Delta/\alpha \prec: q, \Gamma\}\!\} & \widehat{=} & \{\!\{^p/_\alpha\}\!\}\{\!\{\Delta/\Gamma\}\!\} \\ \{\!\{p, \Delta/\alpha :\succ q, \Gamma\}\!\} & \widehat{=} & \{\!\{^p/_\alpha\}\!\}\{\!\{\Delta/\Gamma\}\!\} \end{array}$$

where ϵ is the empty substitution. Otherwise $\{\!\{^\Delta/_\Gamma\}\!\}$ is undefined.

The rule (Subst New) is the only significant addition to the calculus. It creates a new context inside p. This is captured by finding a fresh variable α' to represent the new context and adding the appropriate constraint to the existing nesting constraints Π_0. The new context is substituted into the expression, which is then evaluated.

The rule (Subst Expose) follows [13] by relying on α-conversion to make the variable bound in the **hide** expression the same as that in the **expose** expression, resulting in a simpler presentation. The rule is otherwise standard.

Contexts do not affect computation and could be erased. They do however record the nesting required to obtain the containment invariant.

Evaluation commences with configuration $(\emptyset, \emptyset, a)$, typed as $\emptyset; K \vdash (\emptyset, \emptyset, a) : A$ against a permission consisting only of constants.

9 Properties

The type system has been proven sound. This result depends on the following fundamental lemma.

$$\frac{E; K \vdash C}{E; K \vdash (C)(\emptyset) \Rightarrow C}\ \text{(Arg Empty)} \qquad \frac{E; K \vdash v : A \quad E; K \vdash (\Theta)(\Delta) \Rightarrow C}{E; K \vdash (A \rightarrow \Theta)(v, \Delta) \Rightarrow C}\ \text{(Arg Val)} \qquad \frac{E; K \vdash B{<:}A \quad E; K \vdash (\Theta\{\!\!\{{}^{B}/_{X}\}\!\!\})(\Delta) \Rightarrow C}{E; K \vdash (\forall(X{<:}A)\Theta)(B, \Delta) \Rightarrow C}\ \text{(Arg Type)}$$

$$\frac{E \vdash q \prec: p \quad E; K \vdash (\Theta\{\!\!\{{}^{q}/_{\alpha}\}\!\!\})(\Delta) \Rightarrow C}{E; K \vdash (\forall(\alpha \prec: p)\Theta)(q, \Delta) \Rightarrow C}\ \text{(Arg Context } \prec:) \qquad \frac{E \vdash p \prec: q \quad E; K \vdash (\Theta\{\!\!\{{}^{q}/_{\alpha}\}\!\!\})(\Delta) \Rightarrow C}{E; K \vdash (\forall(\alpha :\succ p)\Theta)(q, \Delta) \Rightarrow C}\ \text{(Arg Context } :\succ)$$

Figure 12: Well-typed Actual Parameters

$$\frac{E; \langle p \rangle \vdash o : [l_i : \Theta_i{}^{i \in 1..n}]^p_q \quad \iota : [l_i : \Theta_i{}^{i \in 1..n}]^p_q \in E \quad \forall \iota \mapsto o \in \sigma}{E \vdash \sigma}\ \text{(Val Store)} \qquad \frac{E \vdash \sigma \quad E; K \vdash a : A \quad \emptyset \vdash K \quad \mathsf{dom}(\sigma) = \mathsf{dom}(E')}{E; K \vdash (\Pi, \sigma, a) : A}\ \text{(Val Config) (where } E \equiv \Pi, E')$$

Figure 13: Well-typed Stores and Configurations

Lemma 1 *1. If $E; K \vdash v : A$ and $E; K' \vdash A$, then $E; K' \vdash v : A$, where v is a value.*
2. If $E; K \vdash A{<:}B$ and $E; K' \vdash B$, then $E; K' \vdash A{<:}B$.

The first clause in essence states that the type contains enough information to determine the permission required to access a value. This allows values to be passed across an object's boundary whenever the type is well-formed both inside and outside of the object, even though the expression which computed the value may not have been. This was essential for demonstrating type preservation for method select and update.

The second clause states, in effect, that all subtypes of a given type are accessible wherever the type is accessible. This was required for the validity of substitution and subsumption.

Soundness is given in part by the following type preservation result (assuming the usual definition of environment extension, $\gg$):

Theorem 2 (Preservation) *If $E; K \vdash (\Pi, \sigma, a) : A$ and $(\Pi, \sigma, a) \Downarrow (\Pi', \sigma', v)$, then there exists an environment E' such that $E' \gg E$ and $E'; K \vdash (\Pi', \sigma', v) : A$.*

The containment invariant follows from type preservation. Define the *refers to* relation $\rightarrow$ for a store as:

$$\{\iota \rightarrow \iota' \mid \iota \mapsto [l_i = \varsigma(s_i : A_i, \Gamma_i)b_i{}^{i \in 1..n}]^p_q \in \sigma \wedge \iota' \in \mathsf{locs}(b_i)\}$$

where $\mathsf{locs}(b)$ is the set of locations in expression b. The containment invariant states: $\iota \rightarrow \iota' \Rightarrow \mathsf{rep}(\iota) \prec: \mathsf{owner}(\iota')$. The proof that this is indeed invariant is based on the following intuition: A location with owner p is accessible only if permission $\langle p \rangle$ is held. The locations ι' such that $\iota \rightarrow \iota'$ are all accessible using permission $\langle q \uparrow \rangle \cup \lceil \Gamma \rceil$, where q is the representation context of ι, for an appropriate Γ. But the context variables in $\lceil \Gamma \rceil$ cannot be the owner of any location, hence the locations in the method body have owners p' where $q \prec: p'$. From this the desired result follows.

10 Danger: Vampires

The trick to implementing multiple interfaces to an object was to create an object with the same owner as the original object, but with a representation context inside the original object's representation context. Unfortunately, in the raw calculus, this can be ill-used to produce what we call *vampiric* behaviour. The following example illustrates its most brutal form:

$$\begin{array}{l} \mathit{vampire} = \\ \quad \mathbf{expose}\ v\ \mathbf{as}\ \alpha \prec: \epsilon, x : A\ \mathbf{in} \\ \quad\ \ \mathbf{new}\ \beta \prec: \alpha\ \mathbf{in} \\ \quad\ \ \ \ \mathbf{let}\ \mathit{vamp} = [\mathit{drain} = \mathit{drink}(x.\mathit{blood})]^{\epsilon}_{\beta}\ \mathbf{in} \\ \quad\ \ \ \ \ \ \mathbf{hide}\ \beta\ \mathbf{as}\ \beta \prec: \epsilon\ \mathbf{in}\ \mathit{vamp} \end{array}$$

This expression takes a protected object v and unpacks the underlying representation context and unprotected object into α and x, respectively. It then creates a new context β inside α, which becomes the representation context of new object *vamp*. Thus object *vamp* has surreptitious access to the representation of v, without v being aware. Furthermore, because the owner of *vamp* is ϵ, it is accessible anywhere.

This sort of behaviour can be avoided by encoding method select and field update of protected objects in terms of **expose** and then removing **expose** from the programmer's syntax [10].

However, similar vampiric behaviour can occur whenever new contexts are created inside contexts which are passed as method parameters. Classes rely on this idiom. But, in a language based entirely on classes (which does not allow classes to be defined inside methods), this vampiric behaviour is not a problem because the resulting structure of objects is entirely determined by classes, and therefore more tractable to reason about.

Ultimately, this phenomenon means that that representation protection is not as strong as we would like in the raw calculus. It is possible to design syntactic restrictions or encodings which avoid vampiric behaviour, respecting syntactic encapsulation boundaries such as classes and objects.

(Subst Value)

$$\frac{}{(\Pi, \sigma, v) \Downarrow (\Pi, \sigma, v)}$$

(Subst Object) where $o \equiv [l_i = \varsigma(s_i : A_i, \Gamma_i)b_i{}^{i \in 1..n}]^p_q$

$$\frac{\sigma_1 = (\iota \mapsto o) :: \sigma_0 \quad \iota \notin \mathsf{dom}(\sigma_0)}{(\Pi, \sigma_0, o) \Downarrow (\Pi, \sigma_1, \iota)}$$

(Subst Select) where $j \in 1..n$

$$\frac{\sigma_0(\iota) = [l_i = \varsigma(s_i : A_i, \Gamma_i)b_i{}^{i \in 1..n}]^p_q \quad \{\!\{\Delta/\Gamma_j\}\!\} \text{ is defined} \qquad (\Pi_0, \sigma_0, b_j\{\!\{{}^{\iota}/_{s_j}\}\!\}\{\!\{\Delta/\Gamma_j\}\!\}) \Downarrow (\Pi_1, \sigma_1, v)}{(\Pi_0, \sigma_0, \iota.l_j\langle\Delta\rangle) \Downarrow (\Pi_1, \sigma_1, v)}$$

(Subst Update) where $j \in 1..n$

$$\frac{\sigma_0(\iota) = [l_i = \varsigma(s_i : A_i, \Gamma_i)b_i{}^{i \in 1..n}]^p_q \qquad \sigma_1 = \sigma_0 + (\iota \mapsto [l_i = \varsigma(s_i : A_i, \Gamma_i)b_i{}^{i \in 1..j-1, j+1..n}, l_j = \varsigma(s : A_j, \Gamma_j)b]^p_q)}{(\Pi, \sigma_0, \iota.l_j \Leftarrow \varsigma(s : A, \Gamma_j)b) \Downarrow (\Pi, \sigma_1, \iota)}$$

(Subst Let)

$$\frac{(\Pi_0, \sigma_0, a) \Downarrow (\Pi_1, \sigma_1, v) \quad (\Pi_1, \sigma_1, b\{\!\{{}^{v}/_{x}\}\!\}) \Downarrow (\Pi_2, \sigma_2, u)}{(\Pi_0, \sigma_0, \textbf{let } x : A = a \textbf{ in } b) \Downarrow (\Pi_2, \sigma_2, u)}$$

(Subst Unfold)

$$\frac{}{(\Pi, \sigma, \textbf{unfold}(\textbf{fold}(A, v))) \Downarrow (\Pi, \sigma, v)}$$

(Subst New)

$$\frac{\alpha' \notin \mathsf{dom}(\Pi_0) \quad \Pi_1 = (\Pi_0, \alpha' \prec: p) \quad (\Pi_1, \sigma_0, a\{\!\{{}^{\alpha'}/_{\alpha}\}\!\}) \Downarrow (\Pi_2, \sigma_1, v)}{(\Pi_0, \sigma_0, \textbf{new } \alpha \lhd p \textbf{ in } a) \Downarrow (\Pi_2, \sigma_1, v)}$$

(Subst Expose)

$$\frac{(\Pi_0, \sigma_0, b\{\!\{{}^{v}/_{x}\}\!\}\{\!\{{}^{p}/_{\alpha}\}\!\}) \Downarrow (\Pi_1, \sigma_1, u)}{(\Pi_0, \sigma_0, \textbf{expose } (\textbf{hide } p \textbf{ as } \alpha \prec: p' \textbf{ in } v{:}A) \textbf{ as } \alpha \prec: p'', x{:}A' \textbf{ in } b{:}B) \Downarrow (\Pi_1, \sigma_1, u)}$$

Figure 14: Dynamic Semantics

Unfortunately, the type system cannot express or enforce such distinctions.

11 Related Work

Our **new** term which generates new "names" has been studied by Pitts and Stark [29] and independently by Odersky [26] in a different context. It also resembles ν operator in the π calculus and related calculi [23]. These do not include names in types.

Recently Cardelli, Ghelli and Gordon use names at the level of types to prove security properties about the π-calculus [7] and the Ambient calculus [6], and to demonstrate the soundness of the Tofte and Talpin's [32] regions calculus [33]. We believe that this work is complementary, and, in a sense, orthogonal to the work presented here, because they capture properties of ephemeral entities such as references on the stack, rather than the deep structure of objects in the store. A thorough comparison has been slated for future work.

Recently, Gabbay and Pitts introduces a new quantifier over fresh names [14], which is used by Cardelli and Gordon [8] to study the logical properties of name restriction. On the surface, it seems that this is more suited to our calculus for hiding representation contexts than existential quantification. However, it is not entirely appropriate since it precludes methods which return self.

12 Conclusion and Future Work

We modelled object ownership in a small extension to Abadi and Cardelli's object calculus. The result satisfies a particular containment property and offers a framework to study schemes which tame unconstrained aliasing in object-oriented programs. We have made the following contributions beyond our previous work: a sound extension of the object calculus with object ownership, where ownership is separate from objects; and a more flexible containment model obtained by having separate representation and owner contexts.

The main problem remaining, apart from the vampiric action noted in Section 10, is that types may be syntactically too burdensome to use in practice. If the types could be inferred based on some small syntactic specification, such as an annotation indicating which fields are representation, then ownership and containment could find application in future object-oriented programming languages.

Stay tuned for the author's thesis which further explores ownership and containment [10].

Acknowledgements

Many thanks to John Potter and James Noble for their continued support, to Sophia Drossopoulou and Alex Buckley for forcing me to improve my explanations, to Ron van der

Meyden for a few general hints which enabled me to hurdle a major obstacle, and to the anonymous referees for advice which improved this paper.

References

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

[2] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The design and analysis of computer algorithms*. Addison-Wesley, 1974.

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

[4] Boris Bokowski and Jan Vitek. Confined Types. In *OOPSLA Proceedings*, 1999.

[5] Luca Cardelli. *Type Systems*, chapter 103, pages 2208–2236. The Computer Science and Engineering Handbook. Allen B. Tucker (Ed.). CRC Press, 1997.

[6] Luca Cardelli, Giorgio Ghelli, and Andrew D. Gordon. Ambient groups and mobility types. In *Theoretical Computer Science; Exploring New Frontiers in Theoretical Informatics. International Conference IFIP TCS 2000*, volume 1872 of *LNCS*, pages 333–347, 2000.

[7] Luca Cardelli, Giorgio Ghelli, and Andrew D. Gordon. Secrecy and group creation. In *CONCUR 2000 – Concurrency Theory. 11th International Conference*, volume 1877 of *LNCS*, pages 365–379, August 2000.

[8] Luca Cardelli and Andrew D. Gordon. Logical properties of name restriction. In *28th ACM Symposium on Principles of Programming Languages*, January 2001.

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

[10] David G. Clarke. *Ownership and Containment*. PhD thesis, School of Computer Science and Engineering, University of New South Wales, Sydney, Australia, 2001. In preparation.

[11] 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.

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

[13] Cormac Flanagan and Martín Abadi. Types for Safe Locking. In *Programming Languages and Systems*, volume 1567 of *Lecture Notes in Computer Science*, pages 91–107, March 1999.

[14] M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax involving binders. In *14th Annual Symposium on Logic in Computer Science*, pages 214–224. IEEE Computer Society Press, Washington, 1999.

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

[16] A. D. Gordon, P. D. Hankin, and S. B. Lassen. Compilation and equivalence of imperative objects. *Journal of Functional Programming*, 9(4):373–426, July 1999.

[17] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

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

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

[20] 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.

[21] K. Rustan M. Leino. Data Groups: Specifying the Modification of Extended State. In *OOPSLA Proceedings*, 1998.

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

[23] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, Parts I and II. *Information and Computation*, 100:1–77, September 1992.

[24] P. Müller and A. Poetzsch-Heffter. Universes: A type system for controlling representation exposure. In A. Poetzsch-Heffter and J. Meyer, editors, *Programming Languages and Fundamentals of Programming*. Fernuniversität Hagen, 1999.

[25] 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.

[26] Martin Odersky. A functional theory of local names. In *21th ACM conference on Principles of Programming Languages*, January 1994.

[27] Benjamin C. Pierce. Programming with intersection types, union types, and polymorphism. Technical Report CMU-CS-91-106, Carnegie Mellon University, February 1991.

[28] Benjamin C. Pierce and David N. Turner. Statically typed friendly functions via partially abstract types. Technical Report ECS-LFCS-93-256, University of Edinburgh, LFCS, April 1993. Also available as INRIA-Rocquencourt Rapport de Recherche No. 1899.

[29] A. M. Pitts and I. D. B. Stark. Observable properties of higher order functions that dynamically create local names, or: What's new? In *Mathematical Foundations of Computer Science, Proc. 18th Int. Symp., Gdańsk, 1993*, volume 711 of *Lecture Notes in Computer Science*, pages 122–141. Springer-Verlag, Berlin, 1993.

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

[31] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. In *1992 ACM Conference on LISP and Functional Programming*, pages 288–298, San Francisco, CA, June 1992. ACM.

[32] Mads Tofte and Jean-Pierre Talpin. Region-Based Memory Management. *Information and Computation*, 132(2):109–176, 1997.

[33] Silvano Dal Zilio and Andrew D. Gordon. Region analyis and a π-calculus with groups. In *25th International Symposium on Mathematical Foundations of Computer Science, MFCS 2000*, Bratislava, Slovak Republic, August 28–September 1 2000.