Simple Ownership Types for Object Containment

David G. Clarke   James Noble   John M. Potter

November 28, 2000

Abstract

Containment of objects is a natural concept that has been poorly supported in object-oriented programming languages. For a predefined set of ownership contexts, this paper presents a type system that enforces certain containment relationships for run-time objects. A fixed ordering relationship is presumed between the owners.

The formalisation of *ownership types* has developed from our work with *flexible alias protection* together with an investigation of structural properties of object graphs based on dominator trees. Our general ownership type system permits fresh ownership contexts to be created at run-time. Here we present a simplified system in which the ownership contexts are predefined.

Our formal system is presented in an imperative variant of the object calculus. We present type preservation and soundness results. Furthermore we highlight how these type theoretic results establish the maintenance of a containment invariant for objects, in which access to contained objects is only permitted via their owners. In effect, the predefined ownership ordering restricts the permissible inter-object reference structure.

Keywords: OO type systems; ownership types; object containment; flexible alias protection.

1 Introduction

Object-oriented programs suffer from a lack of object-level encapsulation. This gives rise to problems with aliasing of objects, leading, in turn, to difficulties with maintaining correct and robust program behaviours. To cope with difficulties related to complex control flow, structured programming imposes a single input, single output control-flow discipline. This makes it feasible to abstract the program logic using preconditions and postconditions. Structured programming discipline is so common-place nowadays that the benefits of the approach are largely presumed. Unfortunately many of the benefits of the structured approach are lost in object-oriented programming: object aliasing leads to a loss of modularity in reasoning about programs.

We aim to encapsulate objects, imposing structure via object containment, yet retaining most of the flexibility and benefits of object-oriented programming. The key ideas have evolved from our work on *flexible alias protection*

[34], the recognition of implicit structure in object systems [36], and the ability of type systems to impose this structure [14]. In comparison with flexible alias protection, we provide a formal notion of representation containment with accompanying properties. Our earlier ownership type system [14] was based on a Java-like language, but had a number of restrictions. In particular, the containment structure was defined by objects having a fixed and unique owner, thereby forming an ownership tree. An associated *containment invariant* directly captures the idea that the structure of object systems is reflected in the *dominator tree* for the underlying object-reference graph, as we describe elsewhere [36]. The structural constraints imposed by this earlier ownership type system are too rigid to permit them to be used together with some typical object-oriented programming idioms, such as iterators. Furthermore, we did not address subtyping. In this paper we redress some of these limitations.

We introduce an extra degree of freedom into the ownership type system, separating the notion of *contexts* from objects themselves.

In practice, contexts may be associated with static entities such as classes, packages or modules, security domains or network locations. Every object is assigned an *owner* context, which together with the predefined containment ordering, determines which other objects may access it. Objects also have a *representation* context we call simply *rep*; an object's rep determines those objects it may access. To keep things simple, this paper assumes that there is a pre-defined containment ordering on contexts. In other words, we presume the existence of a partial order on a set of fixed contexts, $(\mathcal{C}, \prec:)$. We describe a soundness result and sketch the proof of containment invariance for stored objects. Our containment invariant states that for a reference from an object with identity ι to one with ι' to exist, it is necessary that $\mathsf{rep}(\iota) \prec: \mathsf{owner}(\iota')$.

We think of the owner context as providing a domain for the object's external interface, and the representation context as providing a domain for its implementation. We insist that owners are contained within reps, which is our take on the *no representation exposure* property of flexible alias protection.

The separation of rep and owner context is the key contribution of this paper, resulting in a type system that can model a wide range of alias protection schemes:

- The simplest arangement has one context per object, with contexts forming a tree where every object's rep context is directly inside its owner context. This models full alias encapsulation [34] such as Islands [23] or Balloons [3], providing strong protection against aliasing but restricting programs — for example, forbidding collections of shared objects.

- Similar arangements with coarser granularity can model Sandwich types [17] with one context per class, and Confined Types [8] with one context per Java-style package. Per-class or per-package contexts obviously provide less alias control, but are less intrusive on practical programs.

- Keeping per-object contexts, but utilising the genericity latent in our type systmes allows us to model the containment features of Flexible Alias Pro-

tection [34], and so support collections of shared objects, but not iterators acting directly on those collections representations. This is effectively our earlier Ownership Types system [14], although the earlier work did not support subtyping or inheritance.

- Separating representations and owners by one context models the core of the Universes proposal [30], allowing several iterators to access the representation of another object. Universes itself also includes additional, per-class contexts to model objects owned by static fields (class variables).

- Finally, further seperation of representation and owner contexts allows objects' interfaces to be exported arbirtarily far from their representations in a controlled manner, so that an object's primary interface can belong to some deeply contained subobject. Unlike other systems, this flexibility allows us to model the use of iterators over internal collections as part of a larger abstraction's interface (common in programs using the C++ STL), and COM-style interface aggregation.

We employ a notation which extends the imperative object calculus of Abadi and Cardelli [1] with owner and rep contexts for objects. We chose this notation to simplify the statement and proof of properties; however the essence of the ownership type system should be easy to translate to any object-oriented notation. The key novelty of our type system is its use of permissions (or capabilities). Permissions are sets of contexts. Well-formedness of expressions relies on holding sufficient permission. Owners determine which contexts are needed in the permission, and rep determines which permissions are held.

2 Object Calculus with Contexts

In this section we introduce a syntactic variant of the object calculus which captures the essence of our ownership system. First we outline those aspects of the object calculus that are most relevant for our purposes. Next we motivate our key modifications: one deals with owner and rep decorations for objects; the other imposes a syntactic restriction on the form of method call to prevent object references leaking through method closures. Finally we present the syntax for our variant of the object calculus.

2.1 The Object Calculus

The *Theory of Objects* [1] presents a variety of object calculi of increasing complexity: untyped, first-order typed, imperative, second-order typed, and higher-order typed. We situate our variant somewhere in the middle, choosing an imperative, first-order typed calculus. Being imperative allows us to capture the containment invariant as a property of objects held in the store.

The two basic operations of the object calculus are method select and method update. The untyped functional object calculus, the simplest presented in Chapter 6 of [1], has the following syntax:

$$
\begin{array}{llll}
a, b & ::= & x & \textit{variable} \\
& | & [l_i = \varsigma(x_i)b_i{}^{i \in 1..n}] & \textit{object formation (l_i distinct)} \\
& | & a.l & \textit{field select/method invocation} \\
& | & a.l \Leftarrow \varsigma(x)b & \textit{field update/method update}
\end{array}
$$

An object is a collection of methods labelled l_1 to l_n. Methods take the form $\varsigma(x)b$, where the parameter x refers to the target object, that is *self*, within the method body b. The operational semantics is given by the following reduction rules, where $o \equiv [l_i = \varsigma(x_i)b_i{}^{i \in 1..n}]$:

$$
\begin{array}{llll}
o.l_j & \rightsquigarrow & b_j\{\!\!\{{}^{o}\!/\!_{x_j}\}\!\!\} & (j \in 1..n) \\
o.l_j \Leftarrow \varsigma(x)b & \rightsquigarrow & [l_j = \varsigma(x)b, l_i = \varsigma(x_i)b_i{}^{i \in (1..n) - \{j\}}] & (j \in 1..n)
\end{array}
$$

A method invocation $o.l_j$ reduces to the result of substituting the target object o for the self parameter x_j in the body b_j of the method named l_j. A method update $o.l_j \Leftarrow \varsigma(x)b$ reduces to a copy of the target object where the method in slot l_j is replaced by $\varsigma(x)b$. The definition of reduction is completed by allowing reduction to occur at any subexpression within an expression.

In an imperative object calculus, objects are held in a store which is a map from locations to objects. Evaluation of an object expression amounts to the allocation of that object in the store with a particular object identity denoting its location. Object aliasing arises through sharing of object ids. When a method is invoked, its location (or object id) is substituted for the self parameter. Method update changes the object in-place, rather than constructing a new object as for the functional variants of calculus.

Common object-oriented constructs can easily be captured. Fields are modelled as methods which are values that make no reference to the self parameter. Field update employs method update, again with values making no reference to the self parameter. Functions are modelled as objects where a field is used to store the argument to the function. Methods with arguments are modelled as methods which return a function.

2.2 Extending the Calculus with Contexts

2.2.1 Contexts

The extension of the object calculus presented here is concerned with controlling object access. Adopting the model discussed in the introduction, we modify objects to include both owner and representation contexts. Objects take the form $[l_i = \ldots{}^{i \in 1..n}]_q^p$, where p is the owner and q is the representation context.

We assume that we are given a fixed collection of contexts which form a partial order $(\mathcal{C}, \prec:)$. The relation $\prec:$ is called *inside*. The converse relation $:\succ$ is called *contains*.

Contexts might represent the collection of packages in a Java program. The inside relation $\prec:$ can, for example, represent package nesting. Confined Types [8] uses packages as contexts: each package is associated with two contexts, one confined the other not; the confined version of a package is inside the corresponding unconfined version, and there is no further containment presumed.

A confined type is accessible only through its unconfined version; unconfined types do not restrict access other than via the normal visibility and typing rules for Java. Similarly, contexts could represent a collection of classes where $\prec:$ captures inner class nesting and ϵ again corresponds to some ubiquitous system context. Universes take this approach [30]. Alternatively, contexts could represent some combination of these features, or be based on some other scheme, such as machine names on a network, with $\prec:$ representing subnet relationship.

2.2.2 Evaluation in Context

In our system, certain objects may only be accessible within a particular context. Typically computation proceeds by successive method selection and update on a particular target object. Having access to an object means that access is granted to its representation, but only during the evaluation of a selected method. Unfortunately, the object calculus encodes methods with arguments as methods which return a function closure. The resulting closure can be applied, which is fine, or installed as a part of another object via method update, which is not acceptable from our perspective when the closure contains reference to representation. Thus we need to make a second change to the object calculus to distinguish evaluation which may only occur within a context from values which are accessible from without.

The approach we adopt here is simple. As is common in object-oriented programs we presume that all method calls are fully applied, and that no closures are used. So, unlike the object calculus, we actually use a method call syntax, rather than the syntactically simpler method select. A more complex system that associates evaluation contexts with closures is indeed possible, but we prefer to keep it simple.

Thus we modify objects further so that methods take a pre-specified number of arguments: $[l_i = \varsigma(x_i, \Gamma_i)b_i{}^{i \in 1..n}]_q^p$, where Γ denotes the additional formal parameters. Method select now becomes $o.l_j\langle\Delta\rangle$, where Δ is the actual parameter list. Method update is modified in the obvious manner.

2.3 Syntax for an Object Calculus with Ownership

We now present our variant of the object calculus incorporating ownership.

Figure 1 gives the syntax for *permissions, types, values, objects, expressions, parameters, stores,* and *configurations*. Permissions and contexts were described above. We describe the remainder in turn.

Types Types include just object and method types.

The object type $[l_i : A_i{}^{i \in 1..n}]_q^p$ lists the names and types of the methods of the object, as well the owner context p (superscript) and the representation context q (subscript). Objects of this type can only be accessed in expressions possessing at least permission $\langle p\rangle$. Similarly, this type can only be constructed given the same permission.

$$
\begin{array}{llll}
p & \in & \text{CONTEXT} & = & \mathcal{C} \\
K & \in & \text{PERMISSION} & ::= & \langle p \rangle \\
& & & | & \langle p\!\uparrow\rangle \\[2mm]
A, B, C & \in & \text{TYPE} & ::= & [l_i : A_i{}^{i \in 1..n}]_q^p \\
& & & | & A \mid A \to B \\[2mm]
x & \in & \text{VAR} \\
u, v & \in & \text{VALUE} & ::= & x \\
& & & | & \iota \\
o & \in & \text{OBJECT} & ::= & [l_i = \varsigma(x_i : A_i, \Gamma_i)b_i{}^{i \in 1..n}]_q^p \quad (l_i \ \text{distinct}) \\
a, b & \in & \text{EXPRESSION} & ::= & v \\
& & & | & o \\
& & & | & v.l\langle\Delta\rangle \qquad\qquad \text{where } \Delta ::= \emptyset \mid v, \Delta \\
& & & | & v.l \Leftarrow \varsigma(x : A, \Gamma)b \\
& & & | & \textbf{let } x : A = a \textbf{ in } b \\
\Gamma & \in & \text{PARAM} & ::= & \emptyset \mid x : A, \Gamma \\[2mm]
\sigma & \in & \text{STORE} & ::= & \emptyset \\
& & & | & \sigma, \ \iota \mapsto o \\
s, t & \in & \text{CONFIG} & ::= & (\sigma, a)
\end{array}
$$

Figure 1: The Syntax

Although the form of method type suggests that partial application is permitted, in fact the rule for well-formed method call forces all arguments to be supplied, as discussed above.

Expressions   Expressions are presented in a variant of the *named form* [37]. This amounts to the requirement that the result of (almost) every evaluation step be bound to a variable x which is subsequently used to refer to the result of this computation. While this form does not change the expressiveness of the calculus, it simplifies the statement of its semantics and the proof of its properties.

The language is imperative. Objects evaluate to locations, ι, which are subsequently used to refer to the object in the store. Locations are the only possible result of computation. They do not appear in the expressions a programmer writes.

Objects are given by $[l_i = \varsigma(x_i : A_i, \Gamma_i)b_i{}^{i \in 1..n}]_q^p$. The x_i are the self parameters used in method bodies b_i to refer to the current instance of the object. The type of self A_i is given. The Γ_i are the formal parameters to the method. These are a collection of term variables with their type. The owner context is p and q is the representation context.

Method call $v.l\langle\Delta\rangle$, takes a collection of actual parameters for the method labelled l. The parameters, Δ, are a sequence of values.

Method update, $v.l \Leftarrow \varsigma(x : A, \Gamma)b$, replaces the method labelled l in the object v with that defined by b. Again x is the self parameter and Γ are the formal parameters to the method. As usual, a method can be treated as a field if $x \notin FV(b)$, b is a value, and additionally that $\Gamma = \emptyset$. Thus we do not distinguish between fields and methods.

Let expressions, **let** $x : A = a$ **in** b, are used to define local declarations and to link computations together.

Stores and Configurations   The store, σ, is a map from locations to objects for all locations created throughout the evaluation of an expression. A configuration represents a snapshot of the evaluation. It consists of an expression, a, and a store, σ.

3   Formal Presentation of Ownership Types

3.1   Permissions and Containment

Expressions are typed against a given permission: either a point permission $\langle p \rangle$ representing a singleton context $\{p\}$, or an upset permission $\langle q \uparrow \rangle$ denoting all contexts containing q, that is $\{p \mid q \prec: p\}$.

The point permission $\langle p \rangle$ allows access to objects with owner p. An object with rep q is granted the upset permission $\langle q \uparrow \rangle$, thereby granting it (more precisely, its methods) access to any object with owner in that set.

We write $\iota \rightarrow \iota'$ to indicate that one stored object holds a reference to another. The containment invariant states that:

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

Let $\iota \mapsto [l_i = \varsigma(x_i : A)b_i{}^{i \in 1..n}]_q^p$ be a location-object binding in some store, so that $\mathsf{owner}(\iota) = p$ and $\mathsf{rep}(\iota) = q$. The locations accessible to ι are those appearing in the method bodies b_i. The containment invariant yields an upper bound on these:

$$\{\iota' \mid \mathsf{owner}(\iota') \in \langle q \uparrow \rangle\}. \tag{1}$$

Now consider the object at location ι' which has owner p'. Access to this object requires permission $\langle p' \rangle$, which corresponds to the singleton set $\{p'\}$. Thus ι' is in the set (1) if

$$\langle p' \rangle \subseteq \langle q \uparrow \rangle. \tag{2}$$

This condition is enforced by our type system.

Consider the following simplified version of our object typing rule:

$$\text{(Val Object-Simplified)} \quad (\text{where } A \equiv [l_i : C_i{}^{i \in 1..n}]_q^p)$$
$$\frac{E, s_i : A; \langle q \uparrow \rangle \vdash b_i : C_i \quad \forall i \in 1..n}{E; \langle p \rangle \vdash [l_i = \varsigma(x_i : A)b_i{}^{i \in 1..n}]_q^p : A}$$

The conclusion states, among other things, that the permission required to access this object is $\langle p \rangle$, where p is the owner. 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. But this is exactly the condition (2). Therefore, the only locations accessible in a method body are those permitted by the containment invariant. This property is formally presented in this chapter.

Apart from giving types to the appropriate constructs in the expected manner, the other rules in the type system merely propagate or preserve the constraints we require. Subtyping does not allow the owner information stored in a type to shrink, thus preventing the loss of crucial information. Subtyping was missing from our earlier ownership type system [14]. Additional type rules (Type Allow), (Sub Allow), and part of (Val Subsumption) allow well-formedness for a certain permission to be extended to superset permissions.

The type rules which follow are specified depending on a permission. For the calculus of this paper the minimal required permission can in fact be derived from the types involved, so does not actually need to be explicit. However we adopt the current presentation style, not only because it helps to clarify the role of permissions, but also to cater for extensions such as recursive types, where the permissions can not always be directly inferred.

3.2 Supplementary Notation

The type of an expression depends on a *typing environment*, E, which maps program variables (and locations) to types. The typing environment is organised as a sequence of bindings, where $\emptyset$ denotes the empty environment:

$$E \quad ::= \quad \emptyset \mid E, x : A \mid E, \iota : A$$

The syntax of method formal parameters, Γ, are just a subset of the syntax for environments, which allows them to be treated as environments in the type rules.

The domain of a typing environment $\mathsf{dom}(E)$ extracts the bound in the environment:

$$
\begin{aligned}
\mathsf{dom}(\emptyset) &\ \hat{=}\ \emptyset \\
\mathsf{dom}(E, \iota : A) &\ \hat{=}\ \mathsf{dom}(E) \cup \{\iota\} \\
\mathsf{dom}(E, x : A) &\ \hat{=}\ \mathsf{dom}(E) \cup \{x\}
\end{aligned}
$$

This applies also to Γ.

The *free variables* of an expression, $FV(a)$, are determined by:

$$
\begin{aligned}
FV(x) &\ \hat{=}\ \{x\} \\
FV(\iota) &\ \hat{=}\ \emptyset \\
FV(v.l\langle\Delta\rangle) &\ \hat{=}\ FV(v) \cup FV(\Delta) \\
FV(\emptyset) &\ \hat{=}\ \emptyset \\
FV(v, \Delta) &\ \hat{=}\ FV(v) \cup FV(\Delta) \\
FV(v.l \Leftarrow \varsigma(x : A, \Gamma)b) &\ \hat{=}\ FV(v) \cup FV(\varsigma(x : A, \Gamma)b) \\
FV(\mathbf{let}\ x : A = a\ \mathbf{in}\ b) &\ \hat{=}\ FV(a) \cup (FV(b)\backslash\{x\}) \\
FV([l_i = \varsigma(x_i : A_i, \Gamma_i)b_i{}^{i\in 1..n}]_q^p) &\ \hat{=}\ \bigcup_{i\in 1..n} FV(\varsigma(x_i : A_i, \Gamma_i)b_i) \\
FV(\varsigma(x : A, \Gamma)b) &\ \hat{=}\ FV(b)\backslash(\{x\} \cup \mathsf{dom}(\Gamma))
\end{aligned}
$$

Bound variables are those which appear in a expression but are not free. Substitution rules can be stated in a straightforward manner [1, 22].

3.3   The Type System

We define the type system using nine judgements which are described in Figure 2. Judgements concerning constructs which contain no free variables do not require a typing environment in their specification. All judgements concerned with types and expressions are formulated with respect to a permission K.

To simplify the handling of environments, we employ the notation such as $x : A \in E$ to extract assumptions from the typing environment. Well-formedness of the typing environment E is implicit with such assumptions; that is, $E \vdash \diamond$ is part of the assumption.

$$
\begin{array}{ll}
E \vdash \diamond & \textit{E is well-formed typing environment} \\
K \subseteq K' & \textit{K is a subpermission of } K' \\
K \vdash A & \textit{A is a well-formed type given } K \\
K \vdash A{<}{:}B & \textit{A is a subtype of B given } K \\
E; K \vdash a : A & \textit{a is a well-typed expression of type A in E given } K \\
E; K \vdash (A)(\Delta) \Rightarrow C & \textit{The actual parameters } \Delta \textit{ match method type A} \\
& \quad \textit{with return type C in E given } K \\
E \vdash \sigma & \textit{σ is a well-formed store in E} \\
E; K \vdash (\sigma, a) : A & \textit{(σ, a) is a well-typed configuration with type A} \\
& \quad \textit{in E given } K
\end{array}
$$

Figure 2: Judgements

Well-formed Environments

$$
\text{(Env } \emptyset) \qquad\qquad\qquad \text{(Env } x) \qquad\qquad\qquad\qquad \text{(Env Location)}
$$

$$
\frac{}{\emptyset \vdash \diamond} \qquad
\frac{K \vdash A \quad x \notin \mathsf{dom}(E)}{E, x : A \vdash \diamond} \qquad
\frac{\langle p \rangle \vdash [l_i : A_i{}^{i \in 1..n}]_q^p \quad \iota \notin \mathsf{dom}(E)}{E, \iota : [l_i : A_i{}^{i \in 1..n}]_q^p \vdash \diamond}
$$

Adding a variable to an environment requires that it not be already declared and that its type can be well-formed given some permission (Env x). The permission does not matter at this point, but it will return when x is used in an expression. (Env Location) specifies that locations have object type.

Well-formed Permission and Subpermissions

$$
\text{(SubPerm } p) \qquad \text{(SubPerm } \prec{:}) \qquad \text{(SubPerm Refl)} \qquad\quad \text{(SubPerm Trans)}
$$

$$
\frac{p \in \mathcal{C}}{\langle p \rangle \subseteq \langle p \uparrow \rangle} \qquad
\frac{q \prec: p}{\langle p \uparrow \rangle \subseteq \langle q \uparrow \rangle} \qquad
\frac{}{K \subseteq K} \qquad\quad
\frac{K \subseteq K' \quad K' \subseteq K''}{K \subseteq K''}
$$

All permissions in Permission, that is $\langle p \rangle$ and $\langle p \uparrow \rangle$ for each $p \in \mathcal{C}$, are valid. The subpermission relation $K \subseteq K'$ is defined in the obvious manner given the interpretation of permissions as sets of contexts.

Well-formed Types

$$
\text{(Type Object) } (l_i \text{ distinct}) \qquad\qquad \text{(Type Allow)} \qquad\qquad \text{(Type Arrow)}
$$

$$
\frac{\langle q \uparrow \rangle \vdash \Theta_i \qquad \forall i \in 1..n \quad q \prec: p}{\langle p \rangle \vdash [l_i : \Theta_i{}^{i \in 1..n}]_q^p} \qquad
\frac{K \vdash A \quad K \subseteq K'}{K' \vdash A} \qquad
\frac{K \vdash A \quad K \vdash B}{K \vdash A \to B}
$$

The well-formedness of types depends upon the permission required to access values of that type.

10

The justification for (Type Object) is similar to that for (Val Object) given in Section 3.1. The method types of an object type must be well-formed given permission $\langle q \uparrow \rangle$, where q is the representation context. The permission required to access this type is at least $\langle p \rangle$. The condition $q \prec : p$ implies that $\langle p \rangle \subseteq \langle q \uparrow \rangle$ ensuring that an object can access itself.

The rule (Type Allow) states that well-formedness of types is preserved with extended permissions. Method types resemble function types, as given by (Type Arrow). Interestingly the permission required to form method types only depends on the argument and result types. It does not depend on the method body, which typically will require a larger permission.

Well-formed Subtyping

$$
\begin{array}{c}
\text{(Sub Object)} \quad (l_i \text{ distinct}) \\
\langle q \uparrow \rangle \vdash \Theta_i \quad \forall i \in 1..n+m \quad q \prec : p \\
\hline
\langle p \rangle \vdash [l_i : \Theta_i{}^{i \in 1..n+m}]_q^p <: [l_i : \Theta_i{}^{i \in 1..n}]_q^p
\end{array}
\qquad
\begin{array}{c}
\text{(Sub Allow)} \\
K \vdash A <: B \quad K \subseteq K' \\
\hline
K' \vdash A <: B
\end{array}
$$

The rule (Sub Object) allows object extension, but neither the owner nor the representation context may vary. Subtypes which are valid given some permission are valid with a larger permission, though not necessarily vice-versa, according to rule (Sub Allow).

Well-typed Expressions

$$
\begin{array}{c}
\text{(Val } x) \\
x : A \in E \quad K \vdash A \\
\hline
E; K \vdash x : A
\end{array}
\qquad
\begin{array}{c}
\text{(Val Location)} \\
\iota : [l_i : A_i{}^{i \in 1..n}]_q^p \in E \\
\hline
E; \langle p \rangle \vdash \iota : [l_i : A_i{}^{i \in 1..n}]_q^p
\end{array}
$$

Expressions are typed against a typing environment and a permission. Variable typing is by assumption (Val x), though it requires sufficient permission to construct the type. Locations are similarly typed by assumption, where the permission required is at least the point permission for the owner in location's type (Val Location).

$$
\begin{array}{c}
\text{(Val Object)} \quad (\text{where } A \equiv [l_i : B_i{}^{i \in 1..n}]_q^p \text{ and } B_i \equiv \lfloor \Gamma_i \rfloor_{C_i}) \\
E, s_i : A, \Gamma_i; \langle q \uparrow \rangle \vdash b_i : C_i \quad \forall i \in 1..n \\
\hline
E; \langle p \rangle \vdash [l_i = \varsigma(x_i : A, \Gamma_i) b_i{}^{i \in 1..n}]_q^p : A
\end{array}
$$

The (Val Object) rule requires an auxiliary function $\lfloor \Gamma \rfloor_C$, which converts the method arguments Γ and the return type C into a method type B:

$$
\begin{aligned}
\lfloor \emptyset \rfloor_C &\triangleq C \\
\lfloor x : A, \Gamma \rfloor_C &\triangleq A \to \lfloor \Gamma \rfloor_C
\end{aligned}
$$

11

In (Val Object), each method body b_i of method $\varsigma(x_i : A, \Gamma_i)b_i$ is typed against an environment extended with the self parameter x_i with A, the type of the object being formed, and with the formal parameter list Γ_i. The return type of the method body C_i and the formal parameters are combined to make up the method type B_i. The permission $\langle p \rangle$ is required to create an object with owner context p. The permission the method is typed against is $\langle q\uparrow \rangle$, where q is the representation context. Access to self x_i is permitted because $q \prec: p$ can be inferred from the fact that the object's type A is in environment E.

As an interesting aside, the rule (Val Object) in part performs masking. In general, the permissions used to type an object's methods are different from those required to type the object. The object itself acts as a mask which permits more within the object than without. The effect of the masking becomes apparent when a method is invoked: to invoke a method requires enough permission to access the method and argument types, but this need not be as large as $\langle q\uparrow \rangle$.

$$\text{(Val Select)}$$
$$\frac{E; K \vdash v : [l_i : A_i{}^{i\in 1..n}]_q^p \quad E; K \vdash (A_j)(\Delta) \Rightarrow C_j \quad j \in 1..n}{E; K \vdash v.l_j\langle\Delta\rangle : C_j}$$

(Val Select) requires that the target have an object type with the appropriate method present. The clause $E; K \vdash (A_j)(\Delta) \Rightarrow C_j$ checks that the arguments are well-typed and that access is permitted, and states that the return type is C_j. This form of rule is described below.

$$\text{(Val Update)} \quad (\text{where } A \equiv [l_i : B_i{}^{i\in 1..n}]_q^p \text{ and } B_j \equiv \lfloor\Gamma_j\rfloor_{C_j})$$
$$\frac{E; K \vdash v : A \quad E, x : A, \Gamma_j; K' \vdash b : C_j \quad K' \subseteq \langle q\uparrow \rangle \quad K' \subseteq K \quad j \in 1..n}{E; K \vdash v.l_j \Leftarrow \varsigma(x : A, \Gamma_j)b : A}$$

(Val Update) requires that the target have an object type with the appropriate method present. Firstly, the formal parameter list Γ_j and return type C_j of the new method must be able to be converted to the method's type B_j, that is, $\lfloor\Gamma_j\rfloor_{C_j} = B_j$. The new method body is typed against a self x with the type of the object being updated, and Γ_j. The expression must have type C_j. The given permission K' is, in effect, at most the intersection between the contexts accessible inside the object, $\langle q\uparrow \rangle$, and the accessible contexts in the surrounding expression, K. This 'intersection' of permissions prevents any illegal locations from being added to object, while maintaining the constraints on the expression performing the method update.

$$\text{(Val Let)} \qquad\qquad\qquad\qquad \text{(Val Subsumption)}$$
$$\frac{E; K \vdash a : A \quad E, x : A; K \vdash b : B}{E; K \vdash \mathbf{let}\ x : A = a\ \mathbf{in}\ b : B} \qquad \frac{E; K \vdash a : A \quad K' \vdash A <: B \quad K \subseteq K'}{E; K' \vdash a : B}$$

12

The type rule (Val Let) is standard, except that it carries the same permission through to the expressions.

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

Well-typed Actual Parameter Lists

$$
\begin{array}{cc}
\text{(Arg Empty)} & \text{(Arg Val)} \\[4pt]
\dfrac{E \vdash \diamond \quad K \vdash C}{E; K \vdash (C)(\emptyset) \Rightarrow C} &
\dfrac{E; K \vdash v : A \quad E; K \vdash (B)(\Delta) \Rightarrow C}{E; K \vdash (A \to B)(v, \Delta) \Rightarrow C}
\end{array}
$$

The judgement $E; K \vdash (B_j)(\Delta) \Rightarrow C_j$ guarantees that the actual parameters Δ are correct in number and type, and that the return type is C_j, all with the same permission K.

Well-typed Stores and Configurations

$$
\begin{array}{cl}
\text{(Store Empty)} & \text{(Store Alloc)} \\[4pt]
\dfrac{E \vdash \diamond}{E \vdash \emptyset} &
\dfrac{E \vdash \sigma \quad E; \langle p \rangle \vdash o : [l_i : A_i^{\,i\in 1..n}]_q^p \quad \iota : [l_i : A_i^{\,i\in 1..n}]_q^p \in E}{E \vdash \sigma,\ \iota \mapsto o}
\end{array}
$$

$$
\text{(Val Config)}
$$
$$
\dfrac{E \vdash \sigma \quad E; K \vdash a : A \quad \mathsf{dom}(\sigma) = \mathsf{dom}(E)}{E; K \vdash (\sigma, a) : A}
$$

The objects stored at a location must have the same type as the location. Configuration typing is performed in a typing environment whose domain consists of exactly the locations allocated in the store.

4 Dynamic Semantics and Properties

We now consider an operational semantics, and consequent properties, including type preservation and soundness.

4.1 An Operational Semantics

The operational semantics of the calculus are presented in a big-step, substitution-based style in Figure 3. Fundamentally it differs little from the object calculus semantics of Gordon et. al. [18], though the named form of expression allows some minor simplifications.

13

$$\frac{}{(\sigma, v) \Downarrow (\sigma, v)}$$
(Subst Value)

$$\text{(Subst Object)} \quad \text{where } o \equiv [l_i = \varsigma(x_i : A_i, \Gamma_i)b_i^{\,i \in 1..n}]_q^p$$
$$\frac{\sigma_1 = \sigma_0, \ \iota \mapsto o \quad \iota \notin \mathsf{dom}(\sigma_0)}{(\sigma_0, o) \Downarrow (\sigma_1, \iota)}$$

$$\text{(Subst Select)} \quad \text{where } j \in 1..n$$
$$\frac{\sigma_0(\iota) = [l_i = \varsigma(x_i : A_i, \Gamma_i)b_i^{\,i \in 1..n}]_q^p \quad \{\!\{\Delta/\Gamma_j\}\!\} \text{ is defined}}{\quad (\sigma_0, b_j\{\!\{{}^\iota\!/_{x_j}\}\!\}\{\!\{\Delta/\Gamma_j\}\!\}) \Downarrow (\sigma_1, v) \quad}{(\sigma_0, \iota.l_j\langle\Delta\rangle) \Downarrow (\sigma_1, v)}$$

$$\text{(Subst Update)} \quad \text{where } j \in 1..n$$
$$\sigma_0(\iota) = [l_i = \varsigma(x_i : A_i, \Gamma_i)b_i^{\,i \in 1..n}]_q^p$$
$$\frac{\sigma_1 = \sigma_0 + (\iota \mapsto [l_i = \varsigma(x_i : A_i, \Gamma_i)b_i^{\,i \in 1..j-1, j+1..n}, l_j = \varsigma(x : A_j, \Gamma)b]_q^p)}{(\sigma_0, \iota.l_j \Leftarrow \varsigma(s : A, \Gamma)b) \Downarrow (\sigma_1, \iota)}$$

$$\text{(Subst Let)}$$
$$\frac{(\sigma_0, a) \Downarrow (\sigma_1, v) \quad (\sigma_1, b\{\!\{{}^v\!/_x\}\!\}) \Downarrow (\sigma_2, u)}{(\sigma_0, \mathbf{let}\ x : A = a\ \mathbf{in}\ b) \Downarrow (\sigma_2, u)}$$

Figure 3: Big-step, substitution-based operational semantics

14

The semantics specify an *evaluation relation* between initial and final configurations, $(\sigma, a) \Downarrow (\sigma', v)$. Evaluating expression a with store σ results in a new store σ' and value v.

We use $\sigma, \iota \mapsto o$ to denote extending the store σ with a new location-object binding, where $\iota \notin \mathsf{dom}(\sigma)$. $\sigma + \iota \mapsto o$ denotes updating the store σ so that ι binds to the new object o, where $\iota \in \mathsf{dom}(\sigma)$. In (Subst Select), $b\{\!\{\Delta/\Gamma\}\!\}$, denotes bindings of formal to actual parameters for a method defined as:

$$
\begin{aligned}
\{\!\{\emptyset/\emptyset\}\!\} &\;\hat{=}\; \epsilon \\
\{\!\{v, \Delta / x : A, \Gamma\}\!\} &\;\hat{=}\; \{\!\{{}^v\!/_x\}\!\} \{\!\{\Delta/\Gamma\}\!\}
\end{aligned}
$$

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

Note that only closed terms are evaluated. Evaluation begins with the configuration $(\emptyset, a)$, that is, some closed expression with an empty store. Variables in **let** expressions and method bodies are always bound to values substitutions before they are encountered in evaluation. Expressions either diverge, become stuck, signified by the special configuration WRONG, or result in a value which must be a location.

Firstly, values require no evaluation (Subst Value). Objects evaluate to a new location, which maps to the original object in the new store (Subst Object). The resulting configuration includes the new store. The evaluation rule (Subst Select) binds the actual parameters Δ to the formal parameters Γ within the body of the selected method, and the resulting expression evaluated. (Subst Update) replaces the method named l from the object at location ι with the one supplied. (Subst Let) evaluates the first expression a, substitutes the result for x in b, which is then evaluated.

These rules only apply when all of the underlying assumptions are specified. Error cases are captured by the other evaluation rules which we refrain from exhibiting here. They account for the following errors: message-not-understood error, when the method is not present in the object; an incorrect number of arguments supplied to a method call; and errors propagated through subexpressions.

4.2 Key Properties

The type system is sound. The proofs have been omitted but are generally a straightforward induction.

Soundness depends on the following fundamental lemma.

Lemma 4.1 (Permissibility)

1. *If $E; K \vdash v : A$ and $K' \vdash A$, then $E; K' \vdash v : A$, where v is a value.*

2. *If $K \vdash A{<:}B$ and $K' \vdash B$, then $K' \vdash A{<:}B$.*

The first clause essentially states that the type contains enough information to determine the permission required to access a value. Values can be passed across object boundaries, for example, if the type is well-formed on both the object's inside and outside. This can happen even when the expression computing the value may not have been accessible in both places. This clause is 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 is required for the validity of substitution and subsumption.

Definition 4.2 (Extension) *Environment E' is an extension of E, written $E' \gg E$, if and only E is a subsequence of E'.*

The following type preservation result states that reduction preserves typing:

Theorem 4.3 (Preservation) *If $E; K \vdash (\sigma, a) : A$ and $(\sigma, a) \Downarrow (\sigma', v)$, then there exists an environment E' such that $E' \gg E$ and $E'; K \vdash (\sigma', v) : A$.*

Soundness states that well-typed expressions do not go wrong:

Theorem 4.4 (Soundness) *If $\emptyset; K \vdash (\emptyset, a) : A$ is derivable, then $(\emptyset, a) \not\Downarrow$ WRONG.*

5 The Containment Invariant

The containment invariant is a statement about the well-formedness of stores, in particular the underlying reference structure. The containment invariant states that for well-typed stores the following holds:

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

where $\mathsf{owner}(\iota)$ and $\mathsf{rep}(\iota)$ give the owner and representation context of an object.

To prove this formally requires more work than suggested by the intuition at the start of Section 3.1. The key aspect to enforcing containment is the use of permissions to control object access to the owner contexts. We define a series of owner context functions which collect owner contexts underlying permissions, types, and locations in expressions.

The key results state that the owner context for a value is contained in the underlying contexts for its type, and the owner contexts underlying types and expressions are contained within those of any permission that gives access to the types and expressions. In particular, this means that the permission does really bound the owner contexts of locations in an expression. By applying this result to method bodies — more precisely, the locations a method refers to — we obtain the result we desire. Incidentally, our approach to capturing the containment invariant resembles the *collecting semantics* used in abstract interpretation [31].

The semantic functions are:

16

- $\eta : \textsc{Location} \rightarrow \mathcal{C}$.

- $[\![\_]\!] : \textsc{Permission} \rightarrow \mathbb{P}(\mathcal{C})$.

- $[\![\_]\!] : \textsc{Type} \rightarrow \mathbb{P}(\mathcal{C})$.

- $[\![\_]\!]_\eta : \textsc{Expression} \rightarrow \mathbb{P}(\mathcal{C})$.

We use the notation $\eta \models E$ to state that whenever $\iota : [l_i : A_i{}^{i \in 1..n}]_q^p$ occurs in E, then $\eta(\iota) = p$. That is to say, that η is the same as $\mathsf{owner}(\iota)$. This serves to define η.

The remaining functions are defined as follows:

$$
\begin{aligned}
[\![\langle p \rangle]\!] &\;\hat{=}\; \{p\} \\
[\![\langle q{\uparrow}\rangle]\!] &\;\hat{=}\; \{p \in \mathcal{C} \mid q \prec: p\}
\end{aligned}
$$

$$
[\![[l_i : A_i{}^{i \in 1..n}]_q^p]\!] \;\hat{=}\; \{p\}
$$

$$
\begin{aligned}
[\![x]\!]_\eta &\;\hat{=}\; \emptyset \\
[\![\iota]\!]_\eta &\;\hat{=}\; \{\eta(\iota)\} \\
[\![o]\!]_\eta &\;\hat{=}\; \emptyset \\
[\![v.l\langle \Delta \rangle]\!]_\eta &\;\hat{=}\; [\![v]\!]_\eta \cup [\![\Delta]\!]_\eta \\
\text{where} \quad [\![\emptyset]\!]_\eta &\;\hat{=}\; \emptyset \\
[\![v, \Delta]\!]_\eta &\;\hat{=}\; [\![v]\!]_\eta \cup [\![\Delta]\!]_\eta \\
[\![v.l \Leftarrow \varsigma(x : A, \Gamma)b]\!]_\eta &\;\hat{=}\; [\![v]\!]_\eta \cup [\![b]\!]_\eta \\
[\![\mathbf{let}\ x : A = a\ \mathbf{in}\ b]\!]_\eta &\;\hat{=}\; [\![a]\!]_\eta \cup [\![b]\!]_\eta
\end{aligned}
$$

Note that $[\![a]\!]_\eta$ depends only on the owners of locations.

The following theorem can be proven by mutual deduction on the structure of typing derivations.

Theorem 5.1 *We have the following, where in relevant each case $\eta \models E$,*

1. *If $K' \subseteq K$, then $[\![K']\!] \subseteq [\![K]\!]$;*

2. *If $K \vdash A$, then $[\![A]\!] \subseteq [\![K]\!]$;*

3. *If $K \vdash A{<:}B$, then $[\![A]\!] \subseteq [\![B]\!]$;*

4. *If $E; K \vdash a : A$, then $[\![a]\!]_\eta \subseteq [\![K]\!]$;*

5. *If $E; K \vdash v : A$, then $[\![v]\!]_\eta \subseteq [\![A]\!]$; and*

6. *If $E; K \vdash (\Theta)(\Delta) \Rightarrow C$, then $[\![\Delta]\!]_\eta \subseteq [\![K]\!]$.*

17

It is illuminating to understand the difference between clauses 4 and 5. Clause 4 applies to all expressions including values, stating that permissions control access within expressions. Extending clause 5 to expressions is impossible because an expression may compute using locations which are not captured in the type, yet return a value of a type which only requires a smaller permission to access.

With the notion of a *well-contained store* we capture the containment invariant globally, that is for all stored objects.

Definition 5.2 (Well-contained Store)

$$\boldsymbol{wf}_E(\sigma) \quad \hat{=} \quad \bigwedge_{\iota \mapsto o \in \sigma} \boldsymbol{wf}_E(o)$$

$$\boldsymbol{wf}_E([l_i = \varsigma(x_i : A_i, \Gamma_i)b_i{}^{i \in 1..n}]_q^p) \quad \hat{=} \quad [\![b_i]\!]_\eta \subseteq [\![\langle q \uparrow \rangle]\!] \quad where \ \eta \models E$$

We can easily prove that a well-typed store is well-contained:

Lemma 5.3 *If $E \vdash \sigma$ then $\boldsymbol{wf}_E(\sigma)$.*

We now convert this result to a local one, that is, one defined between pair of locations, thus demonstrating containment invariance. To give a precise definition of the *refers to* relation, we first need to extract the locations present in an expression:

Definition 5.4 (Locations in a Expression) *The set of locations in an expression, $locs(a)$, is defined as follows:*

$$
\begin{aligned}
locs(x) &\quad \hat{=} \quad \emptyset \\
locs(\iota) &\quad \hat{=} \quad \{\iota\} \\
locs([l_i = \varsigma(x_i : A_i, \Gamma_i)b_i{}^{i \in 1..n}]_q^p) &\quad \hat{=} \quad \emptyset \\
locs(a.l\langle \Delta \rangle) &\quad \hat{=} \quad locs(a) \cup locs(\Delta) \\
locs(\emptyset) &\quad \hat{=} \quad \emptyset \\
locs(v, \Delta) &\quad \hat{=} \quad locs(v) \cup locs(\Delta) \\
locs(a.l \Leftarrow \varsigma(x : A, \Gamma)b) &\quad \hat{=} \quad locs(a) \cup locs(b) \\
locs(\boldsymbol{let} \ x : A = a \ \boldsymbol{in} \ b) &\quad \hat{=} \quad locs(a) \cup locs(b)
\end{aligned}
$$

Notice this does not examine inside objects, because objects define a boundary inside of which the permissions may be different. The following Lemma provides the link between the global and local versions of well-containedness:

Lemma 5.5 *If $E; K \vdash a : A$ and $\eta \models E$, then $[\![a]\!]_\eta = \bigcup_{\iota \in locs(a)} [\![\iota]\!]_\eta$.*

The *refers to* relation collects all the locations present in the bodies of all the methods of an object:

Definition 5.6 (*refers to*) *The* refers to *relation,* $\rightarrow_\sigma$*, for store* σ *is defined as:*

$$\iota \rightarrow_\sigma \iota' \ \text{ iff } \ \iota \mapsto [l_i = \varsigma(x_i : A_i, \Gamma_i)b_i{}^{i \in 1..n}]_q^p \in \sigma \wedge \iota' \in \textit{locs}(b_i), \text{ for some } i \in 1..n$$

The functions which give the owner and representation context of an object are defined for a given location-object, $\iota \mapsto [l_i = \varsigma(x_i : A_i, \Gamma_i)b_i{}^{i \in 1..n}]_q^p \in \sigma$, as: $\mathsf{owner}_\sigma(\iota) \ \widehat{=} \ p$ and $\mathsf{rep}_\sigma(\iota) \ \widehat{=} \ q$,

The containment invariant is now straightforward:

Theorem 5.7 (Containment Invariant) *If* $E \vdash \sigma$ *then* $\iota \rightarrow_\sigma \iota' \Rightarrow \mathsf{rep}_\sigma(\iota) \prec: \mathsf{owner}_\sigma(\iota')$.

PROOF:

1. ASSUME: $E \vdash \sigma$ and $\iota \rightarrow_\sigma \iota'$.
2. By Lemma 5.3, $\mathbf{wf}_E(\sigma)$.
3. So $\iota \mapsto [l_i = \varsigma(x_i : A_i, \Gamma_i)b_i{}^{i \in 1..n}]_q^p \in \sigma$, where $\iota' \in \mathsf{locs}(b_i)$ for some i, and $q = \mathsf{rep}_\sigma(\iota)$.
4. By 2 and Definition 5.2, we get $[\![b_i]\!]_\eta \subseteq [\![\langle q \uparrow \rangle]\!]$, where $\eta \models E$.
5. By Lemma 5.5, $[\![\iota']\!]_\eta \subseteq [\![b_i]\!]_\eta$.
6. But $[\![\iota']\!]_\eta = \{\mathsf{owner}_\sigma(\iota)\}$.
7. Therefore $\mathsf{owner}_\sigma(\iota') \in [\![\langle q \uparrow \rangle]\!]$, where $\iota \rightarrow_\sigma \iota'$, and $q = \mathsf{rep}_\sigma(\iota)$.
8. Unravelling the definition of $[\![\langle q \uparrow \rangle]\!]$, we get $\mathsf{rep}_\sigma(\iota) \prec: \mathsf{owner}_\sigma(\iota')$, as required.

6 Related Work

The pointer structures within object-oriented programs have received surprisingly little examination over the last decade — much less interest than has been lavished on the more tractable problem of inheritance relationships between classes. Good surveys of the problems caused (and advantages gained) by aliasing in object-oriented programming can be found elsewhere [24, 3, 34]. More general notions of references, aggregation, object containment, and ownership have been considered in detail, and are now part of accepted standards [12, 19, 26, 16, 36, 35].

Early work [24], beginning with Islands [23] generally took an informal approach to describing or restricting programs' topologies, based on very simple models of containment. Islands, for example, is based on statically checked mode annotations attached to an object's interface, while the more recent Balloons [3] depends upon sophisticated abstract interpretation to enforce the desired restrictions. Often, these kinds of systems mandate copying, swapping, temporary variables, or destructive reads (rather than standard assignments) to protect their invariants, [5, 10, 20, 28, 29, 9], so they are unable to express many common uses of aliasing in object-oriented programs.

Static alias analysis has similarly long been of interest to compiler writers, even since programming languages began to permit aliasing. Whole programs can be analysed directly to detect possible aliasing [27, 15, 25], or hints may

19

be given to the compiler as to probable aliasing invariants [21]. Static analysis techniques have also been developed to to optimise locking in concurrent languages, particularly Java [11, 6, 7, 2, 38]. This work is typically based around escape analysis and removes exclusion from objects that can be shown to be owned by a single thread, although Aldrich et.al.'s work also removes redundant synchronisation due to enclosing locks [2], implying a very simple notion of per-object ownership.

More recent work [33] has attempted to be more practically useful, combining annotations on objects with more flexible models of containment or ownership. Flexible Alias Protection [34] uses a number of annotations to provide nested per-object ownership, while permitting objects to refer to objects belonging to their (transitive) container as well as objects they own directly. A dynamically checked variant has also been proposed [32].

Guava [4] uses a system of annotations on variables and types similar to Flexible Alias Protection, but motivated towards controlling synchronisation in concurrent Java programs, rather than managing aliasing per se. Confined Types [8] uses only one annotation to confine objects inside Java packages. This gives a much coarser granularity of ownership than many other proposals, so is advocated mostly for security reasons. Sandwich Types [17] are similar to Confined Types in that they restrict references from instance of one type to instances of another, however they are intended to improve locality by using a separate heap for each Sandwich. Universes [30] are in some way the most similar to the ownership types we have presented here. Their universes are like our representation contexts, and they do incorporate subtyping. However they do not have the clear separation of owner and rep context that we have presented here.

We have separately developed a more complex ownership type system incorporating dynamic creation of contexts [13]. The work presented in this paper is simpler and closer to proposals such as Confined Types [8]. We believe that the predefined model of containment presented here will offer an appropriate model of containment, particularly for security-sensitive applications.

7   Conclusion

In this paper, we have presented extensions to Abadi and Cardelli's object calculus to describe object ownership. Ownership and representation contexts were added to both objects and object types: the owner context controls which other objects can access an object, while the representation context controls which other objects an object can access. Combined, these form the basis for our containment invariant, which holds for the type system presented here.

The advantage these extensions confer onto the object calculus is simple: the extended calculus can now model containment in a natural and straightforward manner. Due to our static type system, system-level invariants based on containment — on the the object ownership tree — can be described directly and enforced without any runtime overheads.

The simple type system presented in this paper is restricted in that ownership contexts are fixed: new contexts cannot be created at runtime. While this is sufficient to model systems such as Confined Types [8], we are continuing to develop more powerful (and therefore more complex) type systems that can model systems such as Flexible Alias Protection [34] and its even more flexible successors.

References

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

[2] Jonathan Aldrich, Craig Chambers, Emin Gun Sirer, and Susan Eggers. Static analyses for eliminating unnecessary synchronization from Java programs. In *Sixth International Static Analysis Symposium*. Springer-Verlag, September 1999.

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

[4] David F. Bacon, Robert E. Strom, and Ashis Tarafdar. Guava: A dialect of Java without data races. In *OOPSLA'00 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, volume 35, pages 382–400, New York, October 2000. ACM Press.

[5] Henry G. Baker. 'Use-once' variables and linear objects – storage management, reflection and multi-threading. *ACM SIGPLAN Notices*, 30(1), January 1995.

[6] Bruno Blanchet. Escape analysis for object-oriented languages. application to Java. In *OOPSLA Proceedings*, pages 20–34. ACM, 1999.

[7] Jeff Bogda and Urs Hölzle. Removing unnecessary synchronization in Java. In *OOPSLA Proceedings*, pages 35–46. ACM, 1999.

[8] Boris Bokowski and Jan Vitek. Confined Types. In *OOPSLA Proceedings*, 1998.

[9] John Boyland. Alias burying. *Software—Practice & Experience*, 2001. To appear.

[10] Edwin C. Chan, John T. Boyland, and William L. Scherlis. Promises: Limitied specifications for analysis and manipulation. In *IEEE International Conference on Software Engineering (ICSE)*, 1998.

[11] Jong-Deok Choi, M. Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for Java. In *OOPSLA Proceedings*, pages 1–19. ACM, 1999.

[12] Franco Civello. Roles for composite objects in object-oriented analysis and design. In *OOPSLA Proceedings*, 1993.

[13] David Clarke. An object calculus with ownership and containment. In *Foundations of Object-Oriented Languages (FOOL) 2001*, 2001.

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

[15] Alain Deutsch. Interprocedural May-Alias Analysis for Pointers: Beyond k-limiting. In *Proceedigns of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, June 1994.

[16] Jin Song Dong and Roger Duke. Exclusive control within object oriented systems. In *TOOLS Pacific 18*, 1995.

[17] Daniela Genius, Martin Trapp, and Wolf Zimmermann. An approach to improve locality using sandwich types. In *Proceedings of the 2nd Types in Compilation workshop*, number 1473 in Lecture Notes in Computer Science, pages 194–214, Kyoto, Japan, March 1998. Springer-Verlag.

[18] 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.

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

[20] Douglas E. Harms and Bruce W. Weide. Copying and swapping: Influences on the design of reusable software components. *IEEE Transactions on Software Engineering*, 17(5), May 1991.

[21] Laurie J. Hendren and G. R. Gao. Designing programming languages for analyzability: A fresh look at pointer data structures. In *Proceedings of the IEEE 1992 International Conference on Programming Languages*, April 1992.

[22] J. Roger Hindley and Johnathan P. Seldin. *Introduction to Combinators and λ-Calculus*, volume 1. Cambridge University Press, 1986.

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

[24] 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.

[25] Neil D. Jones and Steven Muchnick. Flow analysis and optimization of LISP-like structures. In Steven Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*. Prentice Hall, 1981.

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

[27] William Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4), December 1992.

[28] K. Rustan M. Leino and Raymie Stata. Virginity: A contribution to the specification of object-oriented software. Technical Report SRC-TN-97-001, Digital Systems Research Center, April 1997.

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

[30] 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.

[31] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.

[32] James Noble, David Clarke, and John Potter. Object ownership for dynamic alias protection. In *TOOLS Pacific*, Melbourne, Australia, November 1999.

[33] James Noble, Jan Vitek, and Doug Lea. *Report of the Intercontinental Workshop on Aliasing in Object-Oriented Systems*, volume 1743 of *Lecture Notes in Computer Science*. Springer, Berlin, Heidelberg, New York, 2000.

[34] 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 1998. Springer-Verlag.

[35] John Potter and James Noble. Conglomeration: Realising aliasing protection. In *Proceedings of the Australian Computer Science Conference (ACSC)*, Canberra, January 2000.

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

[37] 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.

[38] John Whaley and Martin Rinard. Compositional pointer and escape analysis for Java programs. In *OOPSLA'99 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, volume 34, pages 187–206, New York, October 1999. ACM Press.