Ownership Types for Flexible Alias Protection

David G. Clarke, John M. Potter, James Noble

Microsoft Research Institute, Macquarie University, Sydney, Australia
{clad,potter,kjx}@mri.mq.edu.au

Abstract

Object-oriented programming languages allow inter-object aliasing. Although necessary to construct linked data structures and networks of interacting objects, aliasing is problematic in that an aggregate object's state can change via an alias to one of its components, without the aggregate being aware of any aliasing.

Ownership types form a static type system that indicates object ownership. This provides a flexible mechanism to limit the visibility of object references and restrict access paths to objects, thus controlling a system's dynamic topology. The type system is shown to be sound, and the specific aliasing properties that a system's object graph satisfies are formulated and proven invariant for well-typed programs.

Keywords

Alias protection, sharing, containment, ownership, representation exposure, programming language design

1 Introduction

Sharing objects through aliasing is both a powerful feature of object-oriented programming and a weakness [24]. Changes to an object potentially affect all objects that refer to it, even though the object being changed may be unaware of the other objects. This complicates reasoning about object-oriented programs; specifically, there is a lack of modularity in reasoning due to the inability to localise object references. Programs become difficult to understand because knowledge of complete program behavior is required, ultimately leading to programs which are difficult to maintain. This paper takes the first steps towards imposing discipline in controlling aliasing, something absent in current programming languages.

Our earlier work with dynamic change detection schemes for aggregate objects [37] developed into the notion of *flexible alias protection*. That work proposed that aggregate objects could be defined from components where potential aliasing amongst the components is statically determined using aliasing modes. In [38], we provided the rationale for our model of flexible alias protection, supported by illustrative examples, and suggested incorporating aliasing modes into programming languages. For flexible alias protection three key properties are required: no representation exposure; no role confusion between entities with different modes; and limited dependence on the mutable state of other objects (not addressed in this paper).

In the current paper, we introduce *ownership types*: static types annotated with context declarations that represent object ownership. For an object-based language, we construct a type system based on ownership types and establish its soundness via a subject reduction theorem. We refine the notion of representation exposure into a *restricted visibility* property which limits the extent of object visibility, and a *representation containment* property which provides a notion of containment based on articulation points [2] (also called dominators [3]). Absence of role confusion is phrased in terms of *role separation*. These properties, in particular the representation containment property, represent structural invariants on the object graphs of well-typed programs which indicate the kinds of aliasing that are not possible.

The paper is organised as follows. Section 2 describes ownership types, object contexts and their intended semantics in terms of object ownership, and considers some illustrative examples in a Java-like language extended with ownership types. Section 3 introduces a core language and formalises a static semantics for it in terms of ownership types. Section 4 describes an operational semantics and, by providing a novel interpretation of ownership types, proves a subject reduction property demonstrating soundness for our ownership type system. Section 5 formalises and proves representation containment and other properties. Section 6 discusses: the model of containment suggested by the articulation point formulation; the correspondence between the work presented here and the flexible alias protection model [38]; and any limitations. Section 7 considers related work, including other proposals for providing alias protection. Section 8 concludes with the current status of our theoretical work and of the prototype implementation based on Pizza, together with future directions.

2 Ownership Types and Object Contexts

Typically in object systems, objects live in a global address space called the heap, or object store. There are no strict constraints on which parts of the object store an object can access, because there are no restrictions on the way ob-

ject references are passed around. This has repercussions when preventing representation exposure for aggregate objects. The components which constitute an aggregate object are considered to be contained within that aggregate, and part of its representation. But, because the object store is global, there is, in general, no way to prevent other objects from accessing that representation. Enforcing the notion of containment with the standard reference semantics is impossible.

The key to solving this problem is to introduce the notion of *object contexts*. Each object owns a context, and is owned by the context that it resides within. Object contexts provide a nested partitioning of the object store, reflecting nesting of objects, that allows us to speak of an object's interior and exterior.

We annotate types with *context declarations* to yield *ownership types*. Two variables having ownership types with different declared contexts cannot refer to the same part of the store, thus cannot be aliases for the same object. Our ownership type system establishes a sensible notion of an object's interior.

We now informally develop these ideas.

A program begins execution in a root context. In a program text this is denoted by the context norep (nobody's representation). Since the root context is global, its objects are considered to be owned by the system, and are potentially accessible to all objects in the system. This provides a mechanism for sharing *value* objects [31, 25] and objects which provide system-wide services. The context norep also represents the standard constraints on references in object-oriented programming languages; that is, none.

When a new object is created, it is alloted a new context which is considered to be inside that object, and whose contents are deemed to be owned by that object. In the program text, this partition is denoted by rep, the partition which holds the representation. We impose a restriction on the use of rep, which is approximately that the only objects that can access the rep partition are the object that owns it *and* other objects inside that partition (or inside partitions of objects inside that partition, and so on), *but* only if they have been given explicit access to the rep partition. In other words, a rep partition is not accessible from outside of its owner. This is an informal statement of what we mean by *representation containment*.

Note that rep is object dependent; it denotes a different context for each object, just as this is different for each object.

Within the definition of an object, that is, within its class, the context parameter owner denotes the object context that owns that object (the one referred to by this). The owner of a new object is not necessarily the owner or the rep context of the object creating it. It follows that owning an object and having a reference to it are *not* necessarily the same. The context parameter owner was not present in [38]. The utility of this extension can be seen in the examples. Further, it is essential in assigning an ownership type to this.

We have other context parameters which allow object context information to be passed around. But we do not treat object contexts as first-class values as this would make it impossible to construct a static type system. Instead, we treat context parameters analogously to the type parameters of a generic class in Eiffel [32] or Pizza [39], or a template in C++ [15] (except that we do not macro-expand to perform type-checking as in C++). Context parameters allow ownership information to be passed from one context into a newly created object. This allows us, for example, to create a container whose links are its hidden representation, but whose contents are owned by an object outside of the container [38]. This is not possible with either Balloons [4] or Islands [23].

These object context parameters (including owner) give our system its flexibility.

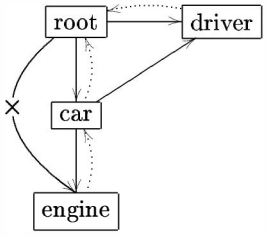
An ownership type consists of a class name, a context representing the owner, and bindings for the context parameters. The bindings to the owner and the context parameters must use the visible contexts within the class where a type is declared, that is rep, norep, owner, plus the context parameters from the class header. Examples of ownership types are given in the following sections.

The discussion here suggests using explicit first-class stores such as those of [35] or the local stores of [46]. But the partitioning of stores is merely a metaphor to enhance understanding. Even the notion of ownership is static; the object stores denoted by object contexts within an object are fixed for the life of that object. There are no operations on object contexts, and their value does not affect computation. Our contexts are statically enforceable, and although they have a run-time interpretation, they do not require run-time support. Thus object contexts and ownership types are static conceptual entities only.

2.1 The object contexts rep and norep

The example in Figure 1 illustrates the object contexts rep and norep.

By executing `Main.main()` until the end of the line marked `(*)`, we get the following object graph:



In this diagram, the solid arrows are references between objects, and the dotted arrows refer to an object's owner. The arrow marked with an × is a reference not allowed in our system.

An `Engine` is part of a `Car` and there should be no external alias to it. Otherwise an engine could be started violating the implied precondition of method `go()`, namely, that a car must have a driver before it can start its engine.

By giving the field `engine` the ownership type `rep Engine`, we are saying that the engine is part of the car's representation and thus it belongs to the context owned by the given instance of `Car`. Furthermore the engine is not accessible outside of this instance.

To outlaw the ×-marked references, we characterise the *representation containment* property as follows: all paths from the root of the system to an object must pass through that object's owner.

Further points to note about this example are as follows.

By typing the field `driver` as `norep Driver`, we consider the car's driver to be owned by the system.

The engine cannot be accessed using the field name, or be returned from an external method call. The first could

```
class Engine {
   void start() { ... }
   void stop() { ... }
}

class Driver { ... }

class Car {
   rep Engine engine;          // representation
   norep Driver driver;        // not representation

   Car() {                     // constructor
      // new engine as part of representation
      engine = new rep Engine();
      driver = null;
   }

   rep Engine getEngine() { return engine; }

   void setEngine(rep Engine e) { engine = e; }

   void go() {
      if (driver != null) engine.start();
   }
}

class Main {
   void main()
   {
      norep Driver bob = new norep Driver();
      norep Car car = new norep Car();

      car.driver = bob;        // (*)
      car.go();
      car.engine.stop();       // fails
      car.getEngine().stop(); // fails
                               //  but not in Java
      rep Engine e = new rep Engine();
      car.setEngine(e);        // fails
                               //  different rep
   }
}
```

Figure 1: Car Example

be prevented in Java by making the engine `private`. But, Java allows the contents of a private field to be returned by a method. This is not allowed for objects with rep-annotated type. For this reason we say that our system provides *object protection*, which is stronger than the name protection offered by conventional programming languages.

The call `car.setEngine(e)` fails because the rep of the caller and the callee refer to different contexts, namely the root and the `car` context, respectively, and are therefore distinct. The two types are incompatible.

The interpretation of rep also implies that the type `rep Engine` will be different in different instances of the class `Car`. This implies that different cars must have different engines.

2.2 Context Parameters

The next example illustrates context parameters, bindings, and some of the restrictions they impose (Figure 2).

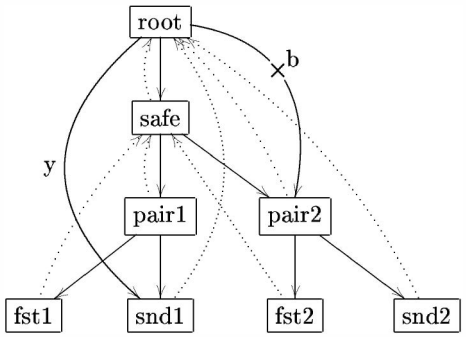
```
class Pair<m,n> {
   m X fst;
   n Y snd;
}

class Intermediate {
   rep   Pair<rep, norep> pair1;
   norep Pair<rep, norep> pair2;

   rep   Pair<rep, norep> a() { return pair1; }
   norep Pair<rep, norep> b() { return pair2; }
   rep   X x() { return pair1.fst; }
   norep Y y() { return pair1.snd; }

   void updateX() {
      pair1.fst = new rep X();
   }
}

class Main {
   norep Intermediate safe;

   void main() {
      rep   Pair<rep, norep> a;
      norep Pair<rep, norep> b;
      rep   X x;
      norep Y y;

      a = safe.a();            // wrong
      b = safe.b();            // wrong
      x = safe.x();            // wrong
      y = safe.y();            // valid

      safe.updateX();          // valid
   }
}
```

Figure 2: Context Parameters Example

The class `Pair` has two context parameters `m` and `n`. By giving `fst` type `m X` and `snd` type `n Y`, we are specifying that the values stored in `fst` and `snd` come from potentially different contexts. The actual context depends upon the bindings to the context parameters `m` and `n` when class `Pair` is used.

Class `Intermediate` uses `Pair` a number of times. The type of the field `pair1` is `rep Pair<rep,norep>` and can be read as follows: the field `pair1` is owned by the current instance; parameter `m` of `Pair` is bound to `rep`, meaning that contents of the field `pair1.fst` are owned by the current instance; and parameter `n` is bound to `norep`, meaning that contents of the field `pair1.snd` are owned by the system.

The following diagram illustrates a typical object graph which this code could represent:



Within `main()` the first three calls to `safe` are invalid because they create alternative access paths to an object's representation. In particular, `b()` does this indirectly by creating a path to `fst2` through `pair2`, circumventing the owner `safe`. Consequently, our type rules must prevent the external visibility of fields whose types use rep in any parameter position.

The call to `y()` is valid because it does not create illegal access paths, since the owner of `snd2` is the root of the system.

From a typing point of view, the invalidity of calls occurs because a type containing rep in one class is different from a syntactically identical type containing rep in a different class.

The call to `updateX()` is valid even though it involves a rep, because the rep in the field `pair1.fst`'s type is the rep from `Intermediate` via a binding to `Pair`'s context parameter `m`. This is as one would expect.

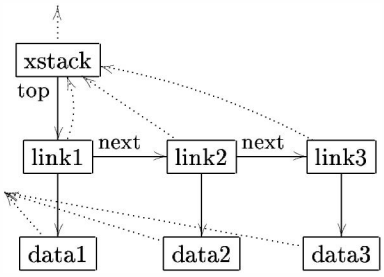
We require textually distinct types appearing in the same class to be distinct, regardless of their bindings. This means, for example, if the field `snd` had type `n X` it would be incompatible with values of type `m X`. This keeps type-checking modular, and also provides a strong notion of *role separation*. The type system guarantees that the two sets of values with different ownership types will be kept separate. In a sense, inside such a class, different virtual partitions are created for each context parameter, even though they may ultimately be unified.

2.3 Owner context

Figure 3 illustrates code for an unbounded stack built from a singly-linked list of `Link` nodes. The class `XStack` provides the handle to the links.

A `Link` node contains a `data` element with context parameter `n`. `XStack` has a context parameter `m` which is bound to `n` of `Link`. This means that the contents of the appropriate `data` elements will be owned by the context corresponding to the binding of the `m` parameter of `XStack`.

An example `XStack` is illustrated in the following diagram:




```
class Link<n> {
   owner Link<n> next;
   n X data;

   Link(n X inData) {
      next = null;
      data = inData;
   }
}

class XStack<m> {
   rep Link<m> top;

   XStack() { top = null; }

   void push(m X data) {
      rep Link<m> newTop = new rep Link<m>(data);
      newTop.next = top;
      top = newTop;
   }

   m X pop() {
      rep Link<m> oldTop = top;
      rep Link<m> top = oldTop.next;
      return top.data;
   }

   boolean isEmpty() { return top == null; }
}
```

Figure 3: Owner Context Example

Without more context we do not know the owners of the data elements (given by the unknown binding to `XStack`'s context parameter `m`) or the owner of the instance `xstack`.

This illustrates the reusability of a class which has context parameters, analogous to generic classes in an object oriented-language with parametric polymorphism.

The main interest in this example is the effect `owner` has on the expressiveness of representation containment. To wit, all the links of the `XStack` are owned by the `xstack` object and are not visible outside of it. Within `XStack`, the links can be freely manipulated. This provides a flexible notion of containment.

To see how this works, observe that the `top` link is rep, so it is owned by the `XStack` instance, `xstack`. Inside the instance of `Link` stored in `top`, owner is bound to the rep context of `xstack`. This can be seen from the diagram. Now the field `top.next` has type `owner Link<m>`. Thus it is also owned by `xstack`. Using the same argument, we can see that `xstack` owns all the links, as shown. Similarly, the data in all the links belongs to the same partition.

It is worth noting that the links of this `xstack` are incompatible with the links of a different instance of `XStack`. The links of a data structure built in this manner are owned exclusively by the handle object (`xstack`). Unlimited self-referential structures can be built using this mechanism.

Another point worth reiterating is that the owner object is not the object that creates another object, nor does the owner have to refer to the objects it owns. The owner merely

forms a boundary between the inside and the outside of an object.

3 Formalisation of an Ownership Type System

The technical part of the paper spans this and the following two sections. We present an object-based language which resembles the core of Java without inheritance, extended with object contexts.

Ownership types are then defined, followed by the usual machinery required to build a type system. The key element in the type system is the *static visibility* constraint present in the type rules for field access and update, and for method call. As these expressions represent object access, we must invalidate those which would give access to another object's representation. Our type system checks each class against the types purported in the class declaration [16]. This keeps the type system simple and modular.

Next we present the operational semantics for our language. The reduction rules are standard, demonstrating that we do *not* require ownership information at run-time. The additional type rules which determine valid stacks and stores are a little different. These type rules are based on *ownership structures* which are novel; they provide a run-time interpretation of ownership types. Ownership structures have the same form as ownership types, but the contexts are replaced by object identifiers for the owner of objects of the given type and the owners of the objects it uses. Using ownership structures, we formulate and prove subject reduction.

Ownership structures also allow us to formulate the desired structural properties of object graphs that our type system enforces, namely, *role separation*, *restricted visibility*, and *representation containment*. After proving these properties, we conclude with a discussion of their impact on aliasing.

3.1 The Core Language

The core language is a simple object-based language with object contexts. This language and the subsequent type rules are based on [16], with our extensions. Unfortunately, our treatment diverges from the concise operational semantics of [16] due to the object dependence of our ownership types.

The core language syntax is:

$$
\begin{array}{rcl}
P & = & \mathit{defn}^*\ \mathit{local}^*\ e \\
\mathit{defn} & = & \textbf{class}\ c\texttt{<}m^*\texttt{>}\ \{\ \mathit{field}^*\ \mathit{meth}^*\ \} \\
\mathit{field} & = & t\ fd \\
\mathit{meth} & = & t\ md\ (\ \mathit{arg}^*\ )\ \{\ \mathit{local}^*\ e\ \} \\
e & = & \textbf{new}\ t \mid \mathsf{null} \mid e;e \mid x \mid x{=}e \\
 & & \mid e.fd \mid e.fd = e \mid e.md(e^*) \\
\mathit{arg} & = & t\ x \\
\mathit{local} & = & t\ y \\
x,y & = & \text{variable name or } \mathsf{this} \\
c & = & \text{a class name} \\
m & = & \text{an object context parameter} \\
M & = & m \mid \mathsf{rep} \mid \mathsf{norep} \mid \Theta \\
fd & = & \text{a field name} \\
md & = & \text{a method name} \\
t & = & c\texttt{<}M|M^*\texttt{>}
\end{array}
$$

We represent a program, P, as a collection of classes followed by an expression to evaluate, together with its local variables.

Classes take a number of context parameters, and consist of a collection of fields and methods. Simple control constructs can be added without affecting the type rules.

Expressions, e, are, in order, object creation, null, sequencing, local variable access, local variable update, field access, field update and method call. We treat method arguments and this as local variables, with the obvious restriction that this cannot be updated.

Types, t, are written $c\texttt{<}M|M^*\texttt{>}$, where c is the underlying class. The parameter preceding the | is the owner, and the remaining parameters are the bindings to the context parameters for class c. This notation differs slightly from the examples: the type previously written `rep c<m,n>` is written $c\texttt{<}\mathsf{rep}|m,n\texttt{>}$. Furthermore, we use Θ as a shorthand to denote owner, and treat it as we treat other context parameters.

The expression **new** $c\texttt{<}M|\tilde{M}\texttt{>}$ can be understood as follows: firstly it creates a new object from class c in context M; it then passes M and $\tilde{M}$ into the new object, so that the context information is available within the new object. For simplicity there are no explicit constructors.

Although Flexible Alias Protection suggested piggybacking context parameters onto type parameters in an object-oriented language with parametric polymorphism [38], to understand the essence of ownership types, we do not include parametric polymorphism in the core language.

3.2 Ownership Types

Before defining ownership types, we introduce *ownership schemes* as a notational convenience. An ownership scheme is a template from which we can create ownership types and, later, ownership structures. Ownership schemes are denoted $\hat{t} \equiv c\texttt{<}\Theta|\tilde{m}\texttt{>}$, where $\tilde{m}$ are the context parameters taken from the declaration of c and Θ is the implicit owner context.

We create an ownership type from an ownership scheme by substituting for the parameters Θ and $\tilde{m}$. Ownership types appear inside class bodies as the declared types of fields, of method arguments and return values, and of local variables. The substituted values are taken from rep, norep, Θ, and the set of context parameters from the declaration of the class in which the ownership type appears.

For example, assume class d is declared as `class d<p1,p2>`. Then it has ownership scheme $d\texttt{<}\Theta|p_1,p_2\texttt{>}$. Let $\Sigma = \{\Theta, p_1, p_2\}$. Then $t = c\texttt{<}p|\tilde{p}\texttt{>}$, where $\{p\} \cup \tilde{p} \subseteq \Sigma \cup \{\mathsf{rep}, \mathsf{norep}\}$, is a valid ownership type appearing in the body of class d. This is captured by the following definition, where Σ will be known from the context.

Definition 1 (Ownership Type) *Let Σ be a set of context parameters and $\hat{t} = c\texttt{<}\Theta|\tilde{m}\texttt{>}$ is an ownership scheme. An* ownership type *is given by substituting elements of $\Sigma \cup \{\mathsf{rep}, \mathsf{norep}\}$ for the parameters of $\hat{t}$.*

3.3 Static Visibility Constraint

The key to the type system is the *static visibility* constraint. It restricts the access to fields and methods which are declared with rep in their type. Such fields and methods contain or access the representation of a particular object. We want to restrict access to those fields and methods to the object that corresponds to the owner of the representation. In a given class, the only expression statically guaranteed

to denote the object which owns the rep is this. Other expressions denote potentially different objects, so the rep in the type of their fields and methods may be different. Based on these observations, we formulate our static visibility constraint as follows:

Definition 2 (Static Visibility) *For an expression e and an ownership type t, we say that t is visible to e if the following condition is satisfied:*

$$SV(e,t) \stackrel{\text{def}}{=} e \neq \mathsf{this} \Rightarrow \mathsf{rep} \notin \mathsf{contexts}(t).$$

$SV(e,t)$ is called the static visibility constraint, *where the function* contexts(_) *extracts the arguments from an ownership scheme, or a structure built from an ownership scheme using a substitution. It is defined as follows:*

$$\mathsf{contexts}(c\langle m|\tilde{m}\rangle) \stackrel{\text{def}}{=} \{m\} \cup \tilde{m}$$

Because we restrict access to "this" which can denote only one object in any context, we say that we provide *object-based protection.*

Note that the static visibility check is performed on the declared type. This allows one to bind rep to the context parameters of internal containers, preventing those containers from being accessible outside of the current object, but allowing representation to be stored within them.

3.4 The Type System

Before the rules of the type system are presented, we define substitution and related functions which appear explicitly in our type system.

3.4.1 Substitution

A *substitution* is a function from context parameters to some set T. Substitutions are *applied* to both ownership schemes and ownership types. The set T will depend on the context in which the substitutions are used.

If $c\langle\Theta|\tilde{m}\rangle$ is an ownership scheme, then a substitution σ is a map from $\{\Theta\} \cup \tilde{m}$ to T. Substitutions are be applied as follows:

$$\begin{aligned}
\sigma(d\langle n|n_1,\ldots,n_n\rangle) &\stackrel{\text{def}}{=} d\langle\sigma(n)|\sigma(n_1),\ldots,\sigma(n_r)\rangle \\
\sigma(n) &\stackrel{\text{def}}{=} p, \qquad \text{if } n \mapsto p \in \sigma \\
\sigma(n) &\stackrel{\text{def}}{=} n, \qquad \text{if } n \notin \mathrm{dom}(\sigma)
\end{aligned}$$

For example, if $\sigma = \{\Theta \mapsto p, m_1 \mapsto p_1, \ldots, m_r \mapsto p_r\}$ is a substitution based on ownership scheme $c\langle\Theta|\tilde{m}\rangle$, then

$$\begin{aligned}
\sigma(d\langle\Theta|m_1,\ldots,m_n\rangle) &= d\langle p|p_1,\ldots,p_n\rangle \\
\sigma(d\langle m_3|\mathsf{rep},\Theta,\mathsf{norep}\rangle) &= d\langle p_3|\mathsf{rep},p,\mathsf{norep}\rangle.
\end{aligned}$$

Allied with substitutions is the function, ψ, which is a kind of inverse to substitution. ψ returns the substitution which was performed on an ownership scheme to generate some other structure, and is defined as:

$\psi(c\langle n|n_1,\ldots,n_r\rangle) \stackrel{\text{def}}{=} \{\Theta \mapsto n, m_1 \mapsto n_1, \ldots, m_r \mapsto n_r\}$
where $\hat{t} = c\langle\Theta|\tilde{m}\rangle$ is the ownership scheme for class c.
By definition, if $\sigma = \psi(t)$ and $\hat{t}$ is the ownership scheme underlying t, then $\sigma(\hat{t}) = t$.

3.4.2 Notation

To simplify the presentation of the type system, we compress each class c in program P into a field and method dictionary as follows:

$\mathcal{A}^c \stackrel{\text{def}}{=} \{fd \mapsto t\}^*$
a map from field names to their types

$\mathcal{M}^c \stackrel{\text{def}}{=} \{md \mapsto \langle(\tilde{t} \longrightarrow t_{res}), \tilde{x}, e, \{y \mapsto t\}^*\rangle\}^*$
a map from method names to a 4-tuple consisting of the method's type (with argument types $\tilde{t}$ and result type t_{res}), its argument variable names, its body, and a mapping from local variables to their types.

These dictionaries are easily determined from a program.

Since the underlying class of an ownership scheme is determined by inspection, we allow the above dictionaries to be associated with ownership schemes, types and structures as well.

3.4.3 The Type System

The type system is defined with respect to a program, P, and two environments: an object context environment and a type environment. An *object context environment*, denoted Σ, is a collection of context parameters, defined as follows:

$$\Sigma ::= \epsilon \mid \Sigma, m$$

where m are context parameters and ϵ is the empty environment. Generally these are the context parameters that appear in a class declaration along with Θ.

A *type environment*, denoted Γ, is a mapping from variables to ownership types, defined as follows:

$$\Gamma ::= \epsilon \mid \Gamma, x : t$$

In our system, the domain of Γ will always include this and appropriate arguments and local variables for checking a method definition. The notation $\Gamma\{\tilde{x} : \tilde{t}\}$ represents extension of a type environment with multiple variables and their types.

The type system is given in Figure 4, and is based on the following judgements.

$\vdash_{\mathsf{p}} P : t$	P is well-formed with ownership type t
$P \vdash_{\mathsf{d}} defn$	*defn* is well-formed
$P, \Sigma, \Gamma \vdash_{\mathsf{m}} meth$	*meth* is well-formed
$P, \Sigma, \Gamma \vdash_{\mathsf{e}} e : t$	e is well-formed with ownership type t
$P, \Sigma \vdash_{\mathsf{t}} t$	t is well-formed

3.4.4 Explanation

The rule (Program) states that a program is valid if each of its definitions are valid, and the expression to be evaluated is well-typed given the types of its local variables. The local variables' types do not contain any context parameters as the expression is not evaluated within the body of some class.

Class validity is checked by the rule (Class). It assigns this the type $c\langle\Theta|\tilde{m}\rangle$. Methods are typed in an environment with this given that type, and Σ includes Θ and the context parameters from the class header. We also require that the field types are well-formed using the same Σ.

(Program)

$$\frac{P \vdash_{\mathsf{d}} defn_j \text{ for } j \in [1,n] \quad P, \emptyset \vdash_{\mathsf{t}} t'_l \text{ for } l \in [1,m] \quad P, \emptyset, \{\tilde{y} : \tilde{t'}\} \vdash_{\mathsf{e}} e : t}{\vdash_{\mathsf{p}} P : t}$$

where $P = defn_1, \ldots, defn_n \; t'_1 \; y_1, \ldots, t'_m \; y_m \; e$

(Class)

$$\frac{P, \Sigma \vdash_{\mathsf{t}} t_j \text{ for } j \in [1,n] \quad P, \Sigma, \{\mathsf{this} : c\texttt{<}\Theta|\tilde{m}\texttt{>}\} \vdash_{\mathsf{m}} meth_k \text{ for } k \in [1,p]}{P \vdash_{\mathsf{d}} \textbf{class } c\texttt{<}\tilde{m}\texttt{>}\{t_1 \; fd_1 \ldots t_n \; fd_n \; meth_1 \ldots \; meth_p\}}$$

where $\Sigma = \{\Theta\} \cup \tilde{m}$

(Type)

$$\frac{M, \tilde{M} \in \Sigma \cup \{\mathsf{rep}, \mathsf{norep}\}}{P, \Sigma \vdash_{\mathsf{t}} c\texttt{<}M|\tilde{M}\texttt{>}}$$

(Method)

$$\frac{P, \Sigma \vdash_{\mathsf{t}} t \quad P, \Sigma \vdash_{\mathsf{t}} t_j \; j \in [1,n] \quad P, \Sigma \vdash_{\mathsf{t}} t'_l \text{ for } l \in [1,m] \quad P, \Sigma, \Gamma\{\tilde{x} : \tilde{t}, \tilde{y} : \tilde{t'}\} \vdash_{\mathsf{e}} e : t}{P, \Sigma, \Gamma \vdash_{\mathsf{m}} t \; md(t_1 \; x_1, \ldots, t_n \; x_n)\{t'_1 \; y_1, \ldots, t'_m \; y_m \;\; e\}}$$

(New)

$$\frac{P, \Sigma \vdash_{\mathsf{t}} t}{P, \Sigma, \Gamma \vdash_{\mathsf{e}} \textbf{new } t : t}$$

(Null)

$$\frac{P, \Sigma \vdash_{\mathsf{t}} t}{P, \Sigma, \Gamma \vdash_{\mathsf{e}} \mathsf{null} : t}$$

(Sequence)

$$\frac{P, \Sigma, \Gamma \vdash_{\mathsf{e}} e_1 : t_1 \quad P, \Sigma, \Gamma \vdash_{\mathsf{e}} e_2 : t_2}{P, \Sigma, \Gamma \vdash_{\mathsf{e}} e_1; e_2 : t_2}$$

(Local Access)

$$\frac{x \in dom(\Gamma)}{P, \Sigma, \Gamma \vdash_{\mathsf{e}} x : \Gamma(x)}$$

(Local Update)

$$\frac{P, \Sigma, \Gamma \vdash_{\mathsf{e}} e : \Gamma(x) \quad x \in \mathrm{dom}(\Gamma) \not/ \{\mathsf{this}\}}{P, \Sigma, \Gamma \vdash_{\mathsf{e}} x = e : \Gamma(x)}$$

(Field Access)

$$\frac{P, \Sigma, \Gamma \vdash_{\mathsf{e}} e : t \quad \sigma = \psi(t) \quad \mathcal{A}^t(fd) = t' \quad SV(e, t')}{P, \Sigma, \Gamma \vdash_{\mathsf{e}} e.fd : \sigma(t')}$$

(Field Update)

$$\frac{P, \Sigma, \Gamma \vdash_{\mathsf{e}} e : t \quad \sigma = \psi(t) \quad \mathcal{A}^t(fd) = t' \quad P, \Sigma, \Gamma \vdash_{\mathsf{e}} e' : \sigma(t') \quad SV(e, t')}{P, \Sigma, \Gamma \vdash_{\mathsf{e}} e.fd = e' : \sigma(t')}$$

(Method Call)

$$\frac{\begin{array}{c} P, \Sigma, \Gamma \vdash_{\mathsf{e}} e : t \qquad \sigma = \psi(t) \\ \mathcal{M}^t(md) = \langle (\tilde{t} \longrightarrow t'), \_, \_, \_ \rangle \\ P, \Sigma, \Gamma \vdash_{\mathsf{e}} e_i : \sigma(t_i) \;\; i \in [1,n] \quad SV(e, t') \wedge \bigwedge_{i \in [1,n]} SV(e, t_i) \end{array}}{P, \Sigma, \Gamma \vdash_{\mathsf{e}} e.md(e_1, \ldots, e_n) : \sigma(t')}$$

Figure 4: The Type System

(Type) uses ownership schemes extracted from the program P to build valid types by performing a substitution where the target set consists of object contexts rep and norep, plus the context parameters in scope (Σ).

(Method) checks the body of the method in the type environment resulting from adding the argument and local variable types to Γ. The rule also requires that the type of each argument and local variable is well-formed with respect to Σ.

(New) creates objects at an ownership type. Similarly, we have a null at each type. This is a restriction because a new object (and null) should be considered *free* [38], and be subsequently coerced to another type. But the notion of free requires a linearity restriction, that is, the new object cannot be shared until we know its correct ownership type, and its integration is a subject for future research. Note that *free* is not the same as *unique* [23], because *free* only applies to newly created objects (and null).

(Sequence) enforces that the type of two expressions in sequence is the type of the latter expression. Local variables are given the type declared in the type environment Γ (Local Access). Rule (Local Update) enforces that this is not updatable, and requires the expression to have the same type as the local variable.

In the rules for (Field Access), (Field Update), and (Method Call), we initially extract from the type of the left hand side expression a substitution, σ. The type of the field/method is extracted from the appropriate dictionary. This type is the type declared in the program text. By applying the substitution to that type, we check the remainder of the expression and give the type of the entire expression. The step would be implicit in a standard formulation of a type system. But, to properly check static visibility and separate reps from different contexts, we need the declared type to perform the $SV()$ check.

Expressions of syntactically different type appearing in the same context are incompatible. This is because there is no subtyping between object contexts, nor at the type level. Because type coercions lose information or require recovery of unknown information, inheritance and subtyping are not yet supported. These and other extensions are the focus of active research.

4 Operational Semantics

This section presents a complete operational model for the core language in terms of reduction rules. These rules describe expression evaluation with respect to a state consisting of a stack and a store. The only surprising feature of the operational semantics is that the object contexts play no role, demonstrating that our type system imposes no runtime overhead.

To demonstrate soundness of our type system, we prove subject reduction. To do so we provide type rules specifing valid execution environments, in a manner similar to [1]. We also introduce a run-time interpretation of ownership types. The context-dependent nature of the type system is clearly apparent from this interpretation function.

A few technical lemmas help us to prove subject reduction. The subject reduction result takes the form of a generalised subject reduction theorem, which applies to open expressions with a given state, rather than closed expressions. From this we get the ordinary subject reduction theorem by specifying an initial execution environment in which to execute a program, P. This environment combined with the generalised subject reduction theorem allows us to formulate structural properties which are proven in the next section.

4.1 Stacks and Stores

The evaluation environment consists of a stack and a store. These are defined as follows:

Definition 3 (The Store) *The* store, $\mathcal{S}$, *is a map from object identifiers, o, to objects, $\mathcal{F}$. An object is a map from field names to values ($v ::= o \mid \mathsf{null}$).*

- $\mathcal{S} \stackrel{\text{def}}{=} \{o \mapsto \mathcal{F}\}^*$ *where* $\mathcal{F} \stackrel{\text{def}}{=} \{fd \mapsto v\}^*$, $\mathrm{dom}(\mathcal{F}) = \mathrm{dom}(\mathcal{A}^c)$, *and c is the class of o.*

We have the following operations on stores:

- $\mathcal{S} \cup \{o \mapsto \mathcal{F}\}$ *is the store obtained by adding to $\mathcal{S}$ a new object o and its field map, where $o \notin \mathrm{dom}(\mathcal{S})$.*
- $\mathcal{S}[o \mapsto \mathcal{F}]$ *is the store obtained from $\mathcal{S}$ by updating object o with field map $\mathcal{F}$, where $o \in \mathrm{dom}(\mathcal{S})$.*
- $\mathcal{F}[fd \mapsto v]$ *is the result of updating field fd of $\mathcal{F}$ with value v.*

Method arguments and local variables are stored on a stack, along with the object whose method is invoked (as variable this). Because stack contents depend on the invoked object, stacks are grouped into stack frames, built from left to right, so that the stack's top is the rightmost stack frame.

Definition 4 (The Stack) *A* stack frame, $\delta \stackrel{\text{def}}{=} \{x \mapsto v\}^*$ *is a map from variables to values. It has the following operations:*

- $\delta(x) \stackrel{\text{def}}{=} v$, *where* $x \mapsto v \in \delta$ (Frame Lookup)
- $\delta[x \mapsto v]$ *is the stack frame formed by replacing the value that x maps to by v* (Frame Update)

A stack, $\Delta \stackrel{\text{def}}{=} \delta_1 \ominus \cdots \ominus \delta_n$, *is a list of stack frames. δ_n is the top of the stack. A stack has the following operations:*

- $\Delta \ominus \delta$ *is the stack constructed by pushing δ onto Δ*
- $\Delta(x) \stackrel{\text{def}}{=} \delta(x)$ (Lookup)
- $\Delta[x \mapsto v] \stackrel{\text{def}}{=} \Delta' \ominus \delta[x \mapsto v]$ (Update)

where $\Delta \equiv \Delta' \ominus \delta$ in the last two operations.

For well-typed programs, all lookups and updates performed in the operational semantics will be on valid variable names. In particular, the variable this exists for all stack frames.

4.2 Reduction Rules

The operational semantics are given in Figure 5. They are expressed in terms of a reduction relation which associates a store $\mathcal{S}$, a stack Δ and a term e with a value v, another store $\mathcal{S}'$, and another stack Δ'. The relation is written:

$$\mathcal{S} \cdot \Delta \vdash e \rightsquigarrow v / \mathcal{S}' \cdot \Delta'$$

(Red New)

$$\frac{E \models S \cdot (\Delta : D) \qquad o \notin \mathrm{dom}(\mathcal{S})}{S \cdot \Delta \vdash \mathbf{new}\ t \rightsquigarrow o / S \cup \{o \mapsto \{fd \mapsto \mathsf{null} \mid fd \in \mathrm{dom}(\mathcal{A}^t)\}\} \cdot \Delta}$$

(Red Local Access)

$$\frac{E \models S \cdot (\Delta : D)}{\mathcal{S} \cdot \Delta \vdash x \rightsquigarrow v / \mathcal{S} \cdot \Delta}$$
where $\Delta(x) = v$

(Red Local Update)

$$\frac{\mathcal{S} \cdot \Delta \vdash e \rightsquigarrow v / \mathcal{S}' \cdot \Delta'}{\mathcal{S} \cdot \Delta \vdash x = e \rightsquigarrow v / \mathcal{S}' \cdot \Delta'[x \mapsto v]}$$

(Red Field Access)

$$\frac{\mathcal{S} \cdot \Delta \vdash e \rightsquigarrow o / \mathcal{S}' \cdot \Delta'}{\mathcal{S} \cdot \Delta \vdash e.fd \rightsquigarrow v / \mathcal{S}' \cdot \Delta'}$$
where $\mathcal{S}'(o)(fd) = v$

(Red Field Update)

$$\frac{\mathcal{S} \cdot \Delta \vdash e \rightsquigarrow o / \mathcal{S}^1 \cdot \Delta^1 \quad \mathcal{S}^1 \cdot \Delta^1 \vdash e' \rightsquigarrow v' / \mathcal{S}^2 \cdot \Delta^2}{\mathcal{S} \cdot \Delta \vdash e.fd = e' \rightsquigarrow v' / \mathcal{S}^2[o \mapsto \mathcal{F}[fd \mapsto v']] \cdot \Delta^2}$$
where $\mathcal{S}^2(o) = \mathcal{F}$

(Red Sequence)

$$\frac{\mathcal{S} \cdot \Delta \vdash e_1 \rightsquigarrow v_1 / \mathcal{S}^1 \cdot \Delta^1 \quad \mathcal{S}^1 \cdot \Delta^1 \vdash e_2 \rightsquigarrow v_2 / \mathcal{S}^2 \cdot \Delta^2}{\mathcal{S} \cdot \Delta \vdash e_1; e_2 \rightsquigarrow v_2 / \mathcal{S}^2 \cdot \Delta^2}$$

(Red Method Call)

$$\frac{\begin{array}{l}\mathcal{S} \cdot \Delta \vdash e \rightsquigarrow o / \mathcal{S}^1 \cdot \Delta^1 \\ \mathcal{S}^1 \cdot \Delta^1 \vdash e_1 \rightsquigarrow v_1 / \mathcal{S}^2 \cdot \Delta^2 \\ \quad \vdots \\ \mathcal{S}^n \cdot \Delta^n \vdash e_n \rightsquigarrow v_n / \mathcal{S}^{n+1} \cdot \Delta^{n+1} \\ \mathcal{S}^{n+1} \cdot \Delta^{n+1} \ominus \{\mathsf{this} \mapsto o, \tilde{x} \mapsto \tilde{v}, \tilde{y} \mapsto \widetilde{\mathsf{null}}\} \vdash e' \rightsquigarrow v' / S' \cdot \Delta^{n+1} \ominus \delta\end{array}}{\mathcal{S} \cdot \Delta \vdash e.md(e_1, \ldots, e_n) \rightsquigarrow v' / \mathcal{S}' \cdot \Delta^{n+1}}$$
where o has type $c\!<\cdot \mid \cdot>$, $\mathcal{M}^c(md) = \langle \_, \tilde{x}, e', l \rangle$,
$\tilde{y} = \mathrm{dom}(l)$ and $\mathrm{dom}(\delta) = \{\mathsf{this}\} \cup \tilde{x} \cup \tilde{y}$

Figure 5: Reduction Rules

and means that with store $\mathcal{S}$ and stack Δ, the term e reduces to value v yielding an updated store $\mathcal{S}'$ and updated stack Δ'. Reductions are in a *natural* or *big-step reduction* style semantics [36].

The reduction rules are standard, unsurprising, and capture the behaviour of a simple object-based language. The assumptions for rules (Red New) and (Red Local Access) are well-typedness of the store and stack. These are defined in Section 4.4. (Red Method Call) requires some explanation: to evaluate the expression $e.md(e_1, \ldots, e_n)$, first evaluate e to some value v; if v is an object o (that is, not null), then each argument e_i is evaluated to some value v_i in the resulting context; a new stack frame is built with this mapped to o, each argument x_i mapped to the appropriate value v_i and each local y_j mapped to null; and then the body of the method is evaluated to v' with the new stack and store. After this evaluation has finished, the top stack frame is discarded. The whole expression evaluates to v'. Note that stack frames lower in the stack are not touched. This last fact can be verified by induction on the rules using the definition of stack.

Note that the rules ignore dereferencing of expressions which evaluate to null. Technically, this would evaluate to some error configuration and additional rules would propogate this through other expressions. Handling null is omitted because it does not change the results and its omission simplifies the presentation.

4.3 Ownership Structures

Ownership structures provide the basis for a semantic interpretation of ownership types. They are used in proving soundness of the type system. The interpretation is novel, but to a certain extent, natural. An object context denotes an owner which is another object, so when interpreting an ownership type, we replace each object context and context parameter by the owner it denotes. This gives an ownership structure, defined as follows:

Definition 5 (Ownership Structure) *Let $\mathcal{O}$ be a set of object identifiers and $c\langle\Theta|\tilde{m}\rangle$ an ownership scheme, then $\tau = c\langle o|\tilde{o}\rangle$, where $\{o\} \cup \tilde{o} \subseteq \mathcal{O}$, is an* ownership structure.

If p is an object whose ownership type is interpreted as $c\langle o|\tilde{o}\rangle$, then the *owner* of p is o. The objects $\tilde{o}$ constitute the owners of objects used by p.

The semantic interpretation, $[\![\_]\!]_o$, is defined for each object, o, and takes ownership types to ownership structures. It uses the ownership structure of o in its definition. This is because any ownership type which $[\![\_]\!]_o$ interprets appears in the body of o's class, c, and is defined in terms of rep, norep, Θ and the context parameters from the class c. An interpretation gives values to each object context and context parameter in scope in class c. The function $[\![\_]\!]_o$ uses the ownership information in o to interpret the types of objects used by o. This provides a way of propogating ownership information from one context to another, which is what the context parameters do statically.

Definition 6 (Semantic Interpretation Function) *Let object o have ownership structure τ, with $c\langle\Theta|\tilde{m}\rangle$ the ownership scheme for τ and $\sigma = \psi(\tau)$. Define $[\![\_]\!]_o$, the* semantic interpretation function for *o, as follows*

$$
\begin{aligned}
[\![\mathsf{rep}]\!]_o &\stackrel{\text{def}}{=} o \\
[\![\Theta]\!]_o &\stackrel{\text{def}}{=} \sigma(\Theta) \\
[\![m]\!]_o &\stackrel{\text{def}}{=} \sigma(m) \\
[\![\mathsf{norep}]\!]_o &\stackrel{\text{def}}{=} \mathsf{root} \\
[\![d\langle n|n_1, \ldots, n_r\rangle]\!]_o &\stackrel{\text{def}}{=} d\langle [\![n]\!]_o | [\![n_1]\!]_o, \ldots, [\![n_r]\!]_o \rangle
\end{aligned}
$$

Translating rep to o confirms the fact that o own's its representation. norep is consistently mapped to root, an object representing the root of the system, which is defined precisely when we specify the initial configuration. The substitution σ allows propogation of the bindings in the context in which o is defined to objects used by o. It must be stressed that rep is interpreted differently for different interpretation functions. Thus, two ownership types containing rep are incompatible precisely when they come from different objects.

It should be clear that not all ownership structures can be built using a given interpretation function. This means that types are not *visible* in all contexts. In particular, types with rep will not be visible outside of their owner. The subject reduction theorem amounts to saying that at no point in a program's execution will an object be in a context where its type is not visible. This notion of visibility is used in defining valid execution environments.

4.4 Valid Execution Environment

We can now precisely define valid stacks and stores, a delicate but essential task for proving subject reduction for imperative languages. This is further complicated by our definition of ownership structures which introduce additional circularity into the definitions: objects are defined in terms of types; types are defined in terms of objects. As we will see, this causes no real difficulty because we can start the definitions using a collection of object identifiers about which nothing is assumed.

Figure 6 provides the complete definition of stacks and store typings. Types are ownership structures built from a set of object identifiers by (Type Type). A store typing, which is a type environment for object identifiers, is a mapping from object identifiers to ownership structures by (Store Type). (Object Type) and (Null Type) correspond to value typings: object types are given in the store typing E, and null can have any available type. (Stack Frame) constrains the construction of valid stack frames so that 1) this is always present, and 2) the type of the other elements in the frame are interpreted correctly with respect to the object whose method was invoked. (Empty Stack) and (Stack Type) build stacks as lists of stack frames. (Store Component) is similar to (Stack Frame); the ownership structure of an object's fields must have an interpretation within that object, where the ownership types are taked from the class definition. (Empty Store) and (Store Construction) build stores from objects. Finally, (Stack Store) defines valid stores and stacks: for some store typing, we require that the store is valid, the stack is valid and that every object assigned a type in E appears in the store.

4.5 Subject Reduction

Soundness of the type system is based on a generalised subject reduction theorem which states that the semantic interpretation of a term's type is invariant under reduction. We prove this theorem for terms which are not closed, having free term variables and context parameters.

o	an object identifier	$v ::= o \mid \mathsf{null}$	a value
$\mathcal{O}$	a set of object identifiers		
$\mathcal{F} ::= \{fd \mapsto v\}^*$	an object	$\mathcal{S} ::= \{o \mapsto \mathcal{F}\}^*$	the store
$\delta ::= \{x \mapsto v\}^*$	a stack frame	$d ::= \{x : \tau\}^*$	stack frame typing
$\Delta ::= \delta_1 \ominus \cdots \ominus \delta_n$	a stack	$D ::= d_1 \ominus \cdots \ominus d_n$	stack typing
$\tau ::= c\langle o \mid o_1, \ldots, o_r \rangle$	an ownership structure	$E ::= \{o \mapsto \tau\}^*$	store typing

$\mathcal{O} \models \tau$	well-formed type judgement
$E \models \diamond$	well-formed store judgement
$E \models v : \tau$	valid typing judgement
$E \models_{sf} \delta : d$	valid stack frame judgement
$E \models_{st} \Delta : D$	valid stack judgement
$E \models o \mapsto \mathcal{F}$	valid store component judgement
$E \models S$	valid store typing judgement
$E \models S \cdot (\Delta : D)$	valid stack and store judgement

(Type Type)
$$\frac{\{o\} \cup \tilde{o} \subseteq \mathcal{O}}{\mathcal{O} \models c\langle o \mid \tilde{o} \rangle}$$

(Store Type)
$$\frac{\mathrm{dom}(E) \models E(o) \quad \text{for each } o \in \mathrm{dom}(E)}{E \models \diamond}$$

(Object Type)
$$\frac{E \models \diamond}{E \models o : E(o)}$$

(Null Type)
$$\frac{E \models \diamond \quad \mathrm{dom}(E) \models \tau}{E \models \mathsf{null} : \tau}$$

(Stack Frame)
$$\frac{E \models o : \tau \quad E \models v_i : [\![t_i]\!]_o \ \text{ for } i \in [1, n]}{E \models_{sf} \{\mathsf{this} \mapsto o, \tilde{x} \mapsto \tilde{v}\} : \{\mathsf{this} : \tau, \tilde{x} : [\![\hat{t}]\!]_o\}}$$
where $\mathsf{contexts}(t_i) \subseteq \mathsf{contexts}(\hat{t}) \cup \{\mathsf{rep}, \mathsf{norep}\}$
and $\hat{t}$ is the ownership scheme of τ

(Empty Stack)
$$\frac{E \models \diamond}{E \models_{st} \emptyset : \emptyset}$$

(Stack Type)
$$\frac{E \models_{st} \Delta : D \quad E \models_{sf} \delta : d}{E \models_{st} \Delta \ominus \delta : D \ominus d}$$

(Store Component)
$$\frac{E \models o : \tau \quad E \models v_i : [\![t_i]\!]_o}{E \models o \mapsto \{fd_i \mapsto v_i\}}$$
where $\tau \equiv c\langle \cdot \mid \cdot \rangle$ and
$fd_i \in \mathrm{dom}(\mathcal{A}^c),\ \mathcal{A}^c(fd_i) = t_i$

(Empty Store)
$$\frac{E \models \diamond}{E \models \emptyset}$$

(Store Construction)
$$\frac{E \models \mathcal{S} \quad E \models o \mapsto \mathcal{F} \quad o \notin \mathrm{dom}(\mathcal{S})}{E \models \mathcal{S} \cup \{o \mapsto \mathcal{F}\}}$$

(Stack Store)
$$\frac{E \models \mathcal{S} \quad E \models_{st} \Delta : D \quad \mathrm{dom}(E) = \mathrm{dom}(\mathcal{S})}{E \models \mathcal{S} \cdot (\Delta : D)}$$

Figure 6: Valid Execution Environment

The full proof of subject reduction is straight-forward but tedious, so has been omitted. Its only enlighting aspect is captured in the following *visibility lemma*, which is the dynamic version of the static visibility constraint. For a given object, we deem an ownership structure to be not visible when the object's interpretation function cannot produce the ownership structure. The visibility lemma captures the notion that if one object, o', has a reference to object, o, then only those ownership types without rep in o can be correctly interpreted by $[\![\_]\!]_{o'}$. So the fields and methods of o that o' can validly access, that is, the ones which are visible, are those without rep in their type, and fields and methods with rep in their type are invisible.

Lemma 1 (Visibility Lemma)
1) If $E \models o : \tau$ then $\tau = [\![\hat{t}]\!]_o$, where $\hat{t}$ is the ownership scheme of τ.
2) Let $E \models o : \tau$ where $\hat{t}$ is the ownership scheme for τ. Let $o' \in \mathrm{dom}(E)$. If $\tau = [\![t]\!]_{o'}$ for some t, then for any ownership type $t^\dagger$ such that $\mathsf{contexts}(t^\dagger) \subseteq \mathsf{contexts}(\hat{t}) \cup \{\mathsf{norep}\}$, we have $[\![\sigma(t^\dagger)]\!]_{o'} = [\![t^\dagger]\!]_o$, where $\sigma = \psi(t)$.

The first clause states that an object has the correct ownership structure with respect to its interpretation function. The second clause states that if the ownership structure of an object can be interpreted with respect to another object's interpretation function, then the second object's interpretation function can interpret any type that the first object's function can, so long as the ownership type does not contain rep. The lemma does not apply at rep, because rep in two different objects is guaranteed to have a different interpretation.

Proof. 1) Holds by definition of $[\![\_]\!]_o$.
2) Let $\Pi = \mathsf{contexts}(\hat{t}) \cup \{\mathsf{norep}\}$. Note that both σ and $[\![\_]\!]_o$ are defined for Π. Firstly, $t = \sigma(\hat{t})$ by the definition of ψ. Given $\hat{t} \equiv c\langle\Theta|\tilde{m}\rangle$ from 1) and the definitions of σ and $[\![\_]\!]_\_$, we have

$$\begin{aligned}[\![\sigma(\hat{t})]\!]_{o'} = [\![\hat{t}]\!]_o &\iff [\![\sigma(c\langle\Theta|\tilde{m}\rangle)]\!]_{o'} = [\![c\langle\Theta|\tilde{m}\rangle]\!]_o \\ &\iff [\![c\langle\sigma(\Theta)|\sigma(\tilde{m})\rangle]\!]_{o'} = [\![c\langle\Theta|\tilde{m}\rangle]\!]_o \\ &\iff c\langle[\![\sigma(\Theta)]\!]_{o'}|[\![\sigma(\tilde{m})]\!]_{o'}\rangle = c\langle[\![\Theta]\!]_o|[\![\tilde{m}]\!]_o\rangle.\end{aligned}$$

Thus, because $\Pi = \{\Theta, \mathsf{norep}\} \cup \tilde{m}$, we have that, for each $m \in \Pi$, $[\![\sigma(m)]\!]_{o'} = [\![m]\!]_o$. (Because norep always maps to root, the equation holds for norep.) Then, by definition of $[\![\_]\!]_\_$, if $\mathsf{contexts}(t^\dagger) \subseteq \Pi$, $[\![\sigma(t^\dagger)]\!]_{o'} = [\![t^\dagger]\!]_o$. □

Because we are evaluating non-closed expressions, we require a condition relating free term variables Γ, with the stack Δ, and free context parameters Σ, with an appropriate interpretation. The condition requires that the execution environment is well typed, that the free variables have an appropriate ownership structure with respect to the current object, $\Delta(\mathsf{this})$, and that the context parameters in the ownership scheme of the current object's type correspond exactly to Σ.

Definition 7 (Valid)
The predicate $\mathsf{valid}(\Sigma, \Gamma, E, \Delta, D)$ is true if and only if $E \models \Delta(\mathsf{this}) : \tau_{\mathsf{this}}$, for some τ_{this}; $\Sigma = \mathsf{contexts}(\hat{t}_{\mathsf{this}})$ where $\hat{t}_{\mathsf{this}}$ is the ownership scheme for τ_{this}; and $\mathrm{dom}(\Gamma) = \mathrm{dom}(d)$ and $\forall x \in \mathrm{dom}(\Gamma)$, $d(x) = [\![\Gamma(x)]\!]_{\Delta(\mathsf{this})}$, where $D = D' \ominus d$.

This validity condition constrains only the top frame of a stack. In the appropriate environment, all other stack frames will be well defined via rules for a validly typed stack, but their value is not used in reduction. This can be demonstrated by an induction on the reduction rules.

The statement of the theorem requires store typing extension. For the full proof, appropriate extension and replacement lemmas hold (analogous to [1]).

Definition 8 (Extension) *We say that E' is an* extension *of E (written $E' \succeq E$) if and only if $\mathrm{dom}(E) \subseteq \mathrm{dom}(E')$ and $\forall o \in \mathrm{dom}(E), E'(o) = E(o)$.*

Note that $\succeq$ is transitive.

Theorem 1 (Generalised Subject Reduction) *If $P, \Sigma, \Gamma \vdash_{\mathsf{e}} e : t$, $\mathcal{S} \cdot \Delta \vdash e \rightsquigarrow v/\mathcal{S}' \cdot \Delta'$, $E \models \mathcal{S} \cdot (\Delta : D)$, and $\mathsf{valid}(\Sigma, \Gamma, E, \Delta, D)$, then there exists an ownership structure τ and a store typing E' such that $E' \succeq E$, $E' \models S' \cdot (\Delta' : D)$, $E' \models v : \tau$, $\tau = [\![t]\!]_{\Delta(\mathsf{this})}$ and $\mathsf{valid}(\Sigma, \Gamma, E', \Delta', D)$.*

Proof. The proof is by induction on the derivation of $\mathcal{S} \cdot \Delta \vdash e \rightsquigarrow v/\mathcal{S}' \cdot \Delta'$, following the style of [1]. The only enlighting aspects of it are captured in the visibility lemma. The static visibility condition invalidates those expressions for which the visibility lemma does not apply, where the representation is accessed outside of its owner. □

4.6 Initial Configuration

To obtain a subject reduction for programs, we need to define an initial configuration in which to begin execution. This configuration is also used in the next section in proving structural properties of object graphs.

Definition 9 (Initial Configuration) *Given a program $P \equiv \widetilde{defn}\ t\ \tilde{y}\ e$, we define the initial store as:*

$$\mathcal{S}_{\mathsf{init}} \stackrel{\mathrm{def}}{=} \{\mathsf{root} \mapsto \emptyset\}.$$

The initial stack is the single stack frame:

$$\delta_{\mathsf{init}} \stackrel{\mathrm{def}}{=} \{\mathsf{this} \mapsto \mathsf{root}, \tilde{y} \mapsto \widetilde{\mathsf{null}}\}.$$

For valid typing, we introduce a class Root *which has no context parameters. The initial store typing $E_{\mathsf{init}} = \{\mathsf{root} : \mathsf{Root}\}$ gives $E_{\mathsf{init}} \models \mathsf{root} : \mathsf{Root}$. Also define $[\![\_]\!]_{\mathsf{root}}$ as*

$$\begin{aligned}[\![\mathsf{rep}]\!]_{\mathsf{root}} &\stackrel{\mathrm{def}}{=} \mathsf{root} \\ [\![\mathsf{norep}]\!]_{\mathsf{root}} &\stackrel{\mathrm{def}}{=} \mathsf{root}.\end{aligned}$$

The object root and the class Root are purely semantic artifacts, because we need an object corresponding to the owner of values with norep as their owner, and for completeness, we also give it a type.

Using the initial configuration above, we get the subject reduction theorem:

Corollary 1
With $P \equiv \widetilde{\mathsf{defn}}\ t\ \tilde{y}\ e$, if $\vdash_{\mathsf{p}} P$, $P, \emptyset, \{\tilde{y} : \tilde{t}\} \vdash e : t$ and $\mathcal{S}_{\mathsf{init}} \cdot \Delta_{\mathsf{init}} \vdash e \rightsquigarrow v/\mathcal{S} \cdot \Delta$, and $E_{\mathsf{init}} \models y_i : [\![t_i]\!]_{\mathsf{root}}$ for each local variable y_i, then there exists an ownership structure τ and a store typing E' such that $E' \succeq E_{\mathsf{init}}$, $E' \models v : \tau$, and $\tau = [\![t]\!]_{\mathsf{root}}$.

5 Structural Properties of Object Graphs

In this section we formulate three structural invariants which hold for well-typed programs. These are *role separation*, *restricted visibility*, and the *representation containment*. The latter has interesting consequences with respect to alias protection and containment, so we devote an additional section to discussing it.

5.1 Role Separation

Role separation is actually a property of the type system. Because there are no coercion operations in the type system and because type checking is modular, we have:

> ROLE SEPARATION: Two different ownership types appearing in the same context are not compatible, regardless of the ensuing bindings.

This means that the type rules are necessarily conservative, but it does guarantee that a class will behave the same way regardless of the bindings to its context parameters. Thus we have a static guarantee that values of two ownership types will not be mixed *within* a class, regardless of their bindings.

5.2 Restricted Visibility

Restricted visibility says dynamic aliasing of representation is not allowed. A dynamic alias is an access which may occur via field access, through the stack, or as the result of a method call [23]. One could imagine such references were valid, even though assignment was not. They aren't.

Recall that we say an object is *visible* in a context if an interpretation exists for its ownership structure. The required property is then:

> RESTRICTED VISIBILITY: Objects assigned to fields, passed as arguments to method calls, returned from field access or method call must be visible in both the context of the callee and the caller.

In particular, one cannot access an object with rep in its type from outside its owner. Restricted visibility follows from subject reduction.

5.3 Representation Containment

Representation containment is our formulation of a no representation exposure property. We characterise this property in terms of paths from the root of the system as follows:

> REPRESENTATION CONTAINMENT PROPERTY: All paths from the root of the system must pass through an object's owner.

Alternatively, the only path from the root of the system to an object's representation is through that object. In a sense, this means that an object is in control of its representation because the rest of the system must use the object's interface to affect any changes upon the representation. And, in a strong sense, the representation can be viewed as being inside its owner.

The remainder of this section is devoted to formalising this in terms of articulation points [2]. The property we prove is actually stronger.

Definition 10 (Articulation Point) *For a graph $\mathcal{G}$, node $k \in \mathcal{G}$ is an* articulation point *for paths from a to b, $a, b \in \mathcal{G}$, if all such paths pass through k.*

Next we define an object graph. This requires a simple modification to a program so that newly created objects appear in the object graph explicitly. This is done simply by requiring that, at the point of creation, a new object is stored in a new local variable of the correct type. It can be shown that this transformation preserves program semantics. (This trick is a simplified version of the *A normal form* [44]).

We use the notation $o \to o'$ to indicate that o has a reference to o', where $o, o' \in \mathrm{dom}(\mathcal{S})$. Intuitively, these are the edges of an object graph:

Definition 11 (Object Graph) *Given a store typing E, a store $\mathcal{S}$ and a stack $\Delta \equiv \delta_1 \ominus \cdot \ominus \delta_n$ such that $E \models \mathcal{S} \cdot (\Delta : D)$ the* object graph, *$\mathcal{G}_{\mathcal{S}\cdot\Delta}$, is the smallest graph satisfying the following:*

1. *for $o = \delta_i(\mathsf{this})$, $i \in [1, n]$, $o \in \mathcal{G}_{\mathcal{S}\cdot\Delta}$* (method invocation target)
2. *for non-*null *$o' \in \mathrm{rng}(\delta_i)$, $o' \in \mathcal{G}_{\mathcal{S}\cdot\Delta}$ and $o \to o' \in \mathcal{G}_{\mathcal{S}\cdot\Delta}$, where $o = \delta_i(\mathsf{this})$, for $i \in [1, n]$* (stack frame reachability)
3. *if $o \in \mathcal{G}_{\mathcal{S}\cdot\Delta}$, with $\mathcal{F} = \mathcal{S}(o)$, then for each non-*null *$o' \in \mathrm{rng}(\mathcal{F})$, $o' \in \mathcal{G}_{\mathcal{S}\cdot\Delta}$ and $o \to o' \in \mathcal{G}_{\mathcal{S}\cdot\Delta}$* (store reachability)

Furthermore, the root *of the graph is $\delta_1(\mathsf{this})$.*

Finally, for an object o, define $\mathsf{ap}(o) \stackrel{\text{def}}{=} \mathsf{contexts}(\tau)$, where τ is the ownership structure of o.

Definition 12 (Representation Containment) *The* representation containment property, *$RC(\_)$ for an object graph, $\mathcal{G}_{\mathcal{S}\cdot\Delta}$ is defined as follows:*

$$RC(\mathcal{G}_{\mathcal{S}\cdot\Delta}) \stackrel{\text{def}}{=} \forall o \in \mathcal{G}_{\mathcal{S}\cdot\Delta},\ \forall k \in \mathsf{ap}(o),\ k \text{ is an articulation point for paths from } \mathsf{root} \text{ to } o.$$

This defines representation containment for a particular snapshot of a program's evolving object graph. It remains to prove that it is invariant over the object graphs for the entire program execution. The argument is inductive in the size of the graph. It is easy to see that removing edges and nodes, or adding nodes does not invalidate the graph, so we just consider adding new edges.

The following lemma provides a necessary condition for a reference to exist.

Lemma 2 *If $\mathcal{T}$ is the derivation tree of the evaluation $\mathcal{S}_{\mathsf{init}} \cdot \Delta_{\mathsf{init}} \vdash e \rightsquigarrow v/\mathcal{S}' \cdot \Delta'$ and $\mathcal{S} \cdot \Delta$ is any store and stack pair appearing in $\mathcal{T}$ (on either side of the evaluation relation), then for all $o \to o' \in \mathcal{G}_{\mathcal{S}\cdot\Delta}$, $\mathsf{ap}(o') \subseteq \mathsf{ap}(o) \cup \{o, \mathsf{root}\}$.*

Proof. Each $o \to o' \in \mathcal{G}_{\mathcal{S}\cdot\Delta}$ exists due to a store or stack reference. Let τ and τ' be the types of o and o'. To be a valid store reference, $\tau' = [\![t']\!]_o$ for some t' by (Store Component). Similary, for stack references by (Stack Frame). By the definition of $[\![\_]\!]_o$, $\mathsf{contexts}(\tau') \subseteq \mathrm{rng}([\![\_]\!]_o) = \mathsf{ap}(o) \cup \{o, \mathsf{root}\}$. $\mathsf{ap}(o') = \mathsf{contexts}(\tau')$, by definition, thus we have $\mathsf{ap}(o') \subseteq \mathsf{ap}(o) \cup \{o, \mathsf{root}\}$. □

The following lemma is standard for articulation points.

Lemma 3 *Let k be an articulation point on paths from r to both p and q in graph $\mathcal{G}$. Adding $p \to q$ to $\mathcal{G}$ preserves the articulation point for both p and q.*

And now the proof that the representation containment property is invariant over the execution of a program:

Theorem 2 (Representation Containment Invariance) *If $\mathcal{T}$ is the derivation tree of the reduction $\mathcal{S}_{\mathsf{init}} \cdot \Delta_{\mathsf{init}} \vdash e \rightsquigarrow v/\mathcal{S}' \cdot \Delta'$ and $\mathcal{S} \cdot \Delta$ is any store and stack pair appearing in $\mathcal{T}$ (on either side of the reduction relation), then $RC(\mathcal{G}_{\mathcal{S}\cdot\Delta})$.*

Proof. 1) Clearly $RC(\mathcal{G}_{\mathcal{S}_{\mathsf{init}}\cdot\Delta_{\mathsf{init}}})$. 2) (Inductive step). Assume that for some well formed $\mathcal{S} \cdot \Delta$, with $\mathcal{G} = \mathcal{G}_{\mathcal{S}\cdot\Delta}$, that $RC(\mathcal{G})$. It is sufficient to consider the effect of adding an edge $p \to n$ (via a valid field update or stack frame construction) to produce a new graph $\mathcal{G}'$. For a contradiction, assume that the representation containment property fails for n in $\mathcal{G}'$.

By Lemma 2, $\mathsf{ap}(n) \subseteq \mathsf{ap}(p) \cup \{p, \mathsf{root}\}$. Otherwise, the assignment would create an invalid store or stack.

Therefore $k \in \mathsf{ap}(p)$ or $k = p$ or $k = \mathsf{root}$.

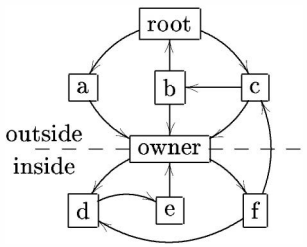
1. $k \in \mathsf{ap}(p)$. Then k is an articulation point for p in $\mathcal{G}$. By lemma 3, k is also an articulation point for p and n in the $\mathcal{G}'$. A contradiction.
2. $k = p$. Removing k from $\mathcal{G}'$ breaks paths from the root containing edge $p \to n$. By assumption, any other path from the root to n had k as an articulation point. Therefore, k remains an articulation point for n. A contradiction.
3. $k = \mathsf{root}$. By removing the root node from $\mathcal{G}$, the articulation property is vacuously true. Again, a contradiction. □

6 Discussion

We now discuss the consequences of basing representation containment on articulation points; we compare this work with flexible alias protection; and we point out some of the limitations.

6.1 Containment

Our model of alias protection is based on restricting the scope of aliasing using containment, rather than linearity constraints which are not entirely compatible with object-oriented programming. By considering the following interpretation of articulation point, we obtain a clear notion of inside and outside. An articulation point partitions a graph into two subgraphs: the *inside*: objects reachable from the root only via paths through the articulation point; and the *outside*: the remainder of the objects. We consider the articulation point, or the owner, to be on the boundary. The following diagram illustrates this idea.



Based on this idea, we can say that flexible alias protection prevents object access from outside its owner. References can go from the inside to the outside, but not vice versa. Our model does not require objects to be referred to by their owner. This allows arbitrary self-referencing data structures to be inside another object. Furthermore, articulation points form a tree (the *dominator tree* in [3]) which corresponds to having nested containment relationships. Elsewhere [40] we have pursued this idea to recognise the inherent containment relationships in evolving object graphs.

Furthermore, the model based on articulation points is faithful to the model of containment in object-oriented modelling given by the *part-of* or *has-a* relationship [43].

6.2 Relationship with Flexible Alias Protection

This work is a formalisation of the core of flexible alias protection [38]. The precise correspondence is discussed in this section.

Flexible alias protection annotated types with mode declarations. These do more than our context declarations by expressing restrictions on effects and dependence on mutable state. Ownership types do not address these issues, as outlined below.

We have introduced the context parameter owner, absent in [38], allowing us to type this. Doing so has also increased our modelling power: we can build an arbitrary self-referencing data structure and impose the restriction that all of its nodes are owned by its handle object. We presented one such example, a stack implemented as a linked list. The context parameter owner adds more flexibility to flexible alias protection.

Flexible alias protection has an aliasing mode var which is described as a "loophole" to obtain normal reference semantics when required. It also has an optional role parameter, which, when present, imposes the restriction that "each role must be kept independent," that we call role separation. In our system, norep corresponds to var without a role parameter, and context parameters correspond to var with a role parameter. The semantics of internal object context rep is consistent with its use in [38].

Finally, our system lacks some elements of the flexible alias protection model. The most significant is arg mode which restricts object access to the interface which does not depend on immutable state. Roughly, this corresponds to parts of an object's extended state which are fixed from the moment the object is initialised. Mode free, omitted for simplicity, represents newly created objects as not being owned by anyone. The mode val for *value* objects was omitted because we are not yet concerned with effects. Value objects can be given any object context, in particular norep, if the values are to be shared arbitrarily. Furthermore, we omitted parametric polymorphism, again for simplicity. Nor do we include inheritance, but [38] makes no mention of it either, due to the difficulty caused by loss of information. Maintaining the representation containment invariant in the presence of subtyping cannot be achieved by extending the system here in the obvious way.

6.3 Limitations

Apart from the object-oriented features, inheritance in particular, not yet incorporated, there are some limitations in the containment model based on articulation points.

Firstly, we cannot, for example, construct external iterators over the representation of some object. An example in [38] solves this by lifting the owner of the representation to the highest context where both the iterator and the list are used. This is clearly unsatisfactory, as it defeats the purpose of containment. The path we intend to take in solving this is by having multiple owners and by reformulating reference containment in terms of *cut sets*. However, we are unsure of how this affects static typing.

The second limitation is that we cannot support multiple representations visible in the same context. An example where this is required is for "friendly functions" such as *plus* for objects representing money, for example. This seems easy to resolve by relaxing the allowable forms of dynamic aliasing.

Finally, change of ownership does not fit well in the model, nor is it easily captured in a static type system.

7 Related Work

The problems engendered by reference semantics and aliasing are well known. These problems are particular severe for object-oriented programming languages [4, 23, 24] where aliasing is common-place.

To control aliasing, researchers have considered explicit notions of aggregation, object containment, and ownership [23, 4, 12, 26, 14, 19]. Hogg's Islands [23] and Almeida's Balloons [4], have a similar aim called *full alias encapsulation* which prohibits any references from crossing the encapsulation boundaries. Unfortunately, full encapsulation of aliasing is too restrictive for many common design idioms used in object-oriented programming, such as sharing data between two collections. Both Islands and Balloons distinguish between static and dynamic aliases — a dynamic alias is caused by a short-lived, stack allocated variable. To gain flexibility, these systems permit various forms of dynamic aliases with otherwise encapsulated objects. Unfortunately, such flexibility often defeats their system's safety. In our system, we protect against both static and dynamic aliases.

The intuition underlying Kent and Maung [26] is at the heart of our work. They proposed an informal extension of the Eiffel programming language with ownership annotations that are to be explicitly tracked and monitored at run-time. Besides statically achieving the constraints their run-time checks impose (apart from their extension for multiple ownership), our system also restricts object visibility through dynamic aliases, something their system fails to do.

None of the above approaches have been completely formalised, despite considerable effort. We have a sound formalisation and proven alias protection guarantees.

Much work has been done in program language semantics in proposing means for coping with pointers and references including [27, 13, 41, 30, 21, 5, 8, 7, 46]. Special reference attachment mechanisms have been proposed for language run-time systems to enforce *unique* or *linear* pointers [6, 11, 34, 20, 28]. Unfortunately, these proposals forbid many common uses of aliasing in object-oriented programs.

A common approach to preventing representation exposure is by restricting the access to variable names. But the *private* and *protected* modes of Java and C++, for example, and related mechanisms [18, 15, 17, 32], or their formal models [42, 1], fail to adequately constrain the sharing of object references. Name protection does not enforce alias protection, because, for example, an externally aliased object can be assigned into a private field, or the contents of a private field can be exported by another means. This is more likely to happen when subclassing a class for which the internal invariants are unknown. Creating external references can break these invariants. Our type system can prevent such external references, thus providing a stronger form of protection than name protection.

In the π-calculus resources are protected by restricting the scope of a name [33]. Other process calculi model *capabilities* [10], which control access to resources. Such notions seem applicable to alias protection. But these calculi are operational models which do not provide static protection. One exception introduces *location types* [22] and provides a static type system which prevents access control to resources for which the capability is not held. But because there is no concrete notion of an object, or object reference, it is unclear how their system translates to object calculi.

The work on security calculi and their type systems are certainly related to our work. Indeed, part of Leroy's formalisation [29] of security properties of strongly-typed applets constrains references in one environment from being accessible in another. To prevent references being leaked to applets, the browsers API must not contain any references types. We see this as a major limitation in his system which the present system overcomes.

The work that most closely resembles ours on a purely formal level is the work on *regions* [45]. Regions are aimed at improving memory management for a functional language (SML), with region information being inferred using Hindley-Milner-style type inference. On the other hand, ownership types are declared and provide alias protection for object-oriented languages. It is nonsensical to infer which objects are representation; such information is part of the programmer's intention. Nevertheless, the underlying techniques are similar, modulo encoding objects into a functional language [9]. To be precise, rep corresponds to letregion, and context parameters correspond to *region variables*. Regions also include read and write *effects*, which we do not.

To our knowledge, no existing formal system statically models object ownership and statically prevents object access from outside an object's owner as ours does.

8 Conclusion

Although necessary in object-oriented programming, unrestricted aliasing is problematic [24], and therefore must be controlled. This paper presents a formalised system providing alias control, proves its soundness and the invariance of specific aliasing properties, namely *role separation*, *restricted visibility* and *representation containment* for well-typed programs. This is the first static type system for an object-based language offering alias protection. Its key mechanisms are limiting visibility of objects using the internal object context rep, and safely extending visibility using context parameters.

We have implemented flexible alias protection by extending Pizza [39]. Our implementation statically validates ownership types according to the type system presented here. With experience, this will allow us to gain a clearer picture

on the use and limitations of ownership types.

There are many directions for future research. Extending the type system to include inheritance is our immediate goal. The control of aliasing with respect to other programming language constructs such as friendly functions and multi-methods will subsequently be considered. Further work will include developing a more thorough understanding of the relationship between the static and dynamic semantics, studying the properties induced by ownership structures, and extending the ideas to concurrent object systems.

Acknowledgements

We would like to thank our colleagues Ryan Shelswell, Geoff Outhred, Ian Joyner, and the anonymous referees for their insight and criticism which made this a better paper. This work was supported by Microsoft Australia.

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] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

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

[5] Pierre America and Frank de Boer. A sound and complete proof system for SPOOL. Technical Report Technical Report 505, Philips Research Laboratories, 1990.

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

[7] Edwin Blake and Steve Cook. On including part hierarchies in object-oriented languages, with an implementation in Smalltalk. In *ECOOP Proceedings*, 1987.

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

[9] Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing Object Encodings. In *Theoretical Aspects of Computer Software (TACS'97)*, LNCS 1281, pages 415–438, 1997.

[10] Luca Cardelli and Andrew D. Gordon. Mobile Ambients. In *Foundations of Software Science and Computation Structures, European Joint Conferences on Theory and Practice of Software*, March 1998.

[11] 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.

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

[13] 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.

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

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

[16] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and Mixins. In *25th ACM conference on Principles of Programming Languages*, January 1998.

[17] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Adison-Wesley, 1983.

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

[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] Matthew Hennessy and James Riely. Resource Control in Systems of Mobile Agents. Technical Report 2/98, University of Sussex, February 1998.

[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] Stuart Kent and John Howse. Value types in Eiffel. In *TOOLS 19*, Paris, 1996.

[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] Xavier Leroy and François Rouaix. Security properies of type applets. In *25th ACM conference on Principles of Programming Languages*, January 1998.

[30] John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *Proceedings of the Eighteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, January 1988.

[31] B. J. MacLennan. Values and objects in programming languages. *ACM SIGPLAN Notices*, 17(12), December 1982.

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

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

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

[35] J. Gregory Morrisett. Refining First-Class Stores. In *ACM SIGPLAN Worshop on State in Programming Languages*, 1993.

[36] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: a formal introduction.* Wiley, 1992.

[37] James Noble and John Potter. Change detection for aggregate objects with aliasing. In *Australian Software Engineering Conference*, Sydney, Australia, 1997. IEEE Press.

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

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

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

[41] John C. Reynolds. Syntatic control of interference. In *5th ACM Symposium on Principles of Programming Languages*, January 1978.

[42] Jon G. Riecke and Chrisopher A. Stone. Privacy via Subsumption. In *Fifth Workshop on Foundations of Object-Oriented Languages*, 1998.

[43] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.

[44] 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.

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

[46] Mark Utting. Reasoning about aliasing. In *The Fourth Australasian Refinement Workshop*, 1995.