Ownership, Encapsulation and the Disjointness of Type and Effect

Dave Clarke
Institute of Information and Computing Sciences
Utrecht University
Utrecht, The Netherlands
dave@cs.uu.nl

Sophia Drossopoulou
Department of Computing
Imperial College
London, Great Britain
sd@doc.ic.ac.uk

ABSTRACT

Ownership types provide a statically enforceable notion of object-level encapsulation. We extend ownership types with computational effects to support reasoning about object-oriented programs. The ensuing system provides both access control and effects reporting. Based on this type system, we codify two formal systems for reasoning about aliasing and the disjointness of computational effects. The first can be used to prove that evaluation of two expressions will never lead to aliases, while the latter can be used to show the non-interference of two expressions.

Categories and Subject Descriptors

D.3.3 [**Software**]: Programming Languages

General Terms

Languages, Theory

Keywords

Ownership types, encapsulation, aliasing, type-and-effects systems.

1. INTRODUCTION

The possibility of aliasing is a key question in many aspects of system development, implementation and optimisation. Compiler optimisations are only valid if the respective manipulations preserve the program's semantics. For example, synchronisation may be eliminated from concurrent programs if it can be established that data races cannot occur [12]. Debugging and code maintenance require knowledge of the aliasing structure and of which parts of an object graph are potentially accessed or changed by the code under consideration [17]. Garbage collection can be assisted when we know that aliases do not, for example, escape beyond a particular region and thus play no further part in computation [8]. Security, such as required for the safe deployment of applets, depends on certain confinement properties [9, 16]. A concrete notion of object aggregate or component can really only exist when the integrity of the encapsulated objects is maintained [33, 22]. Modular reasoning is only possible when some control/knowledge of the aliasing and effects is possible [58, 56, 38, 43].

These tasks require knowledge about potential aliasing and potential accesses to parts of the state. If aliasing and effect information appeared as a part of the type annotations, then the type system could be used to deduce properties of a program's behaviour. These properties are more convenient to obtain than by using a full blown theorem prover, though they can be supplied as axioms to assist the theorem prover [57, 43].

In this paper we suggest the use of ownership types [22, 21, 20] extended with computational effects to address this issue. Ownership types impose structural restrictions on object graphs based on the statically definable and checkable notions of *owner* and of *representation*. The owner of an object is another object and its representation is the objects owned by it. Ultimately, ownership types enforce a statically-checkable notion of encapsulation: every object can have its own private collection of representation objects which are not accessible outside the object which owns them. By adopting ownership as the basis for our effects system, we diverge from the complementary alternative taken by Greenhouse and Boyland [30] who instead employ uniqueness.

In addition to the ownership types and effects system, we introduce two formal systems for reasoning about programs. The first determines which types are disjoint, and can therefore be used to deduce aliasing properties. The second determines which effects statements are disjoint, and can therefore be used to show the non-interference of expressions.

Thus the contributions of this paper are threefold: Firstly, we extend our previous work on ownership types, to support inheritance and dynamic aliases, which allow, *e.g.,* the description of iterators, whilst maintaining a strong notion of encapsulation. Secondly, we extend the ownership types system with computational effects. Thirdly, we apply the ownership types to reason about the absence of aliasing, and apply the effects system to reason about non-interference of computation. We believe that our system is robust, and can be adapted for more sophisticated programming languages.

The paper is structured as follows. Section 2 sets the scene, describing ownership, encapsulation, and how these can be used to reason about the disjointness of type and effect. Section 3 describes $\texttt{Joe}_1$, a core object-oriented language extended with ownership types and effects annotations. In Section 4 we present some examples and discuss some of the consequences of $\texttt{Joe}_1$'s annotations. The static and dynamic semantics of $\texttt{Joe}_1$ are described in Sections 5 and Section 6, along with the statement of relevant properties. Section 7 details our rules for the disjointness of type and effect and illustrates their use. Related work is discussed in Section 8, before we conclude in Section 9, giving also possible directions for future work.

2. OWNERSHIP AND EFFECTS

We describe a notion of object ownership and of encapsulation based on it. We then indicate how these can help determine whether two types are disjoint and whether two effects phrases are disjoint and denote non-interfering computation.

2.1 Contexts, Ownership and Encapsulation

In an ownership types system every object has an owner. The owner is either another object or the predefined constant `world` for objects owned by the system. We call the owners *contexts* and incorporate them into the type system to statically track *object ownership* [22]. Essential differences between object ownership and uniqueness include that an object can refer to objects it does not own but need not refer to objects it does, and that an object may be referred to by any number of objects which do not own it.

In code the owner is indicated by the first parameter of a parameterised type. For example, in Figure 1, the owner of objects in field `head` of class `List` is `this`, the owner the `add` method's argument `d` is `data`, which is a parameter to the `List` class, and the owner of the `Data` object added during the `populate` method is `world`. When no parameters are present, such as for objects of class `Main`, the owner is assumed to be `world`.

From object ownership stems a notion of *containment*: an object is considered to be inside its owner. This induces a transitive, tree-shaped ordering on objects with `world` as the root. By preventing access to an object from objects outside its owner, we gain a strong notion of *encapsulation* which is per object [20]. The exact property, dubbed *owners-as-dominators*, is that objects are dominated by their owners — that is, all paths from the root of the object graph to an object pass through its owner. This has been exhibited for an earlier type system [22] and abstractly in terms of object graphs [49]. A consequence of the owners-as-dominators property is that objects are surrounded by an imaginary boundary which protects internal objects from accesses from external objects.

A system which satisfies the owners-as-dominators property enforces what we call *deep ownership*. The alternative is *shallow ownership* which prevents direct access to the internal objects by ensuring only that their owners are invariant throughout a program's execution, without preventing indirect external access by objects apart from the owner. With shallow ownership, the results presented in this paper would be much weaker. Shallow ownership imposes a weaker constraint on objects graphs, and is thus less restrictive in terms of which programs are legal. We regain some of the potential loss of expressiveness using *dynamic aliases*, stack-based references into the representation of other objects. (In fact, there is really no loss of expressiveness, *per se*. One merely must sacrifice some protection to enable certain implementation patterns.)

```
class List<owner,data> {
  Link<this,data> head;
  void add(Data<data> d) writes under(this) {
    head = new Link<this,data>(d, head);
  }
}
class Main<> {
  List<this,world> list;
  Main() writes this {
    list = new List<this,world>; }
  void populate() writes under(this.1) {
    list.add(new Data<world>);
    list.add(new Data<world>);
  }
  static void main() writes under(world) {
    Main<> main = new Main<>;
    main.populate();
  }
}
```

Figure 1: Example code

Object-level encapsulation is achieved in two steps. Firstly, objects which are owned by the current object (denoted `this`) are given owner context `this`. Such objects are called the *representation*, and the context is called the *representation context*. Access to objects with owner `this` is restricted to the current object and objects inside the current object. Secondly, we ensure that subtyping cannot be used to forget the nesting between objects which would enable references to be passed to objects which ought not have access.

Running the code in Figure 1 results in a two element list with object graph as in Figure 2. The solid circles denote objects. The hollow one, labelled `world`, denotes the top of the system; it need not be an actual object. The solid arrows are ordinary references. The dotted arrows denote object ownership, though these also need not be represented at run-time. Following the owners-as-dominators property, we can depict encapsulation boundaries around objects. These prevent certain references from the outside to the inside, such as the one represented by the dashed arrow.

Starting from the top of the figure, `world` is the pre-existing context for objects owned system-wide. This is the owner of `main` as the class `Main` has no owner parameter. The owner of `data` is also `world` as specified in the type `Data<world>`. The type `List<this,world>` specifies that the list is owned by the instance of `Main`, and that the data in the list is owned by `world`. Additional code (given later in Figure 6) also indicates that all links of the list have the same owner.

2.2 Disjointness of Type and Effect

Using our strong form of encapsulation, we can conclude that certain types are disjoint. For example, if the class `Main` from above had two fields

```
List<this,world>  shared;
List<this,this>   encaps;
```

then we could conclude that `shared` and `encaps` can never be aliases. This is so because `this` and `world` are differ-

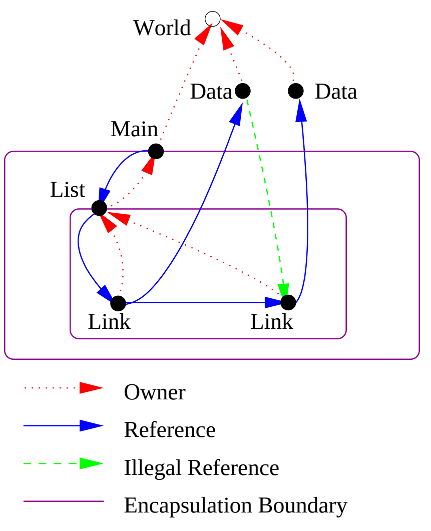

Figure 2: Object Ownership and Encapsulation

ent contexts, since `this` refers to the current instance of `Main` and `world` to a global constant context. Further, we can infer that the `Link` objects referred to by the object in `shared` are disjoint from those referred to by the object in `encaps`, since the collection of links are encapsulated within each list. We can also conclude that the `Data` objects stored in the lists are also disjoint. Viewing `shared` and `encaps` through their common supertype `Object<this>`, however, would not allow anything to be concluded about their alias properties.

We extend the ownership types system from Clarke, Potter and Noble [22] with computational effects. Following Greenhouse and Boyland [30], we allow effects to be calculated for expressions based on declarations which must accompany each method. These declarations enable modular checking.

A primitive effect occurs when the field of an object is read or written. Since contexts correspond to objects, we use contexts as the basis of our effects. An effect is statically denoted as two collections of contexts, called *effect shapes*, containing the contexts which may be read and written. One of the effect shapes denotes all contexts inside and including another. In our example the `add` method of `List` has the write effect `under(this)`. This indicates that a write (or read) may occur to the list object or some object inside it.

The nesting between objects enforced by ownership types allows the description, with some precision, of the effects on unknown parts of an object graph.

For example, we can conclude that running the code

```
shared.add(new Data<world>)
```

will produce an effect which is disjoint from the effect resulting from evaluating

```
encaps.add(new Data<world>).
```

This is because, as we said earlier, `shared` and `encaps` can never be aliases. In addition, because `shared` and `encaps` have the same owner, we can demonstrate that the set of objects denoted by the effect

```
writes under(shared)
```

is disjoint from the set of objects denoted by the effect

```
writes under(encaps).
```

3. JOE = JAVA + OWNERSHIP + EFFECTS

$\texttt{Joe}_1$ is an object-oriented programming language resembling a decaffinated Java extended with both ownership types and effects annotations. The subscript indicates that this is to be the first in a series of languages. The syntax of $\texttt{Joe}_1$ is given in Figure 3.

Notation: $p_{i\in J}$ indicates the sequence (or set) of items p_i for each element i in sequence (or set) J.

Classes.
A class is a collection of fields and methods. In addition, a class may be parameterised by contexts. The first parameter, if present, denotes the owner. Other parameters are used as the owners of objects accessed by objects from this class. Subclassing is specified using an `extends` clause in which the parameters of the superclass are instantiated with contexts in scope, preserving the owner in the first position.

Types.
A type $c\langle p_{i\in 1..n}\rangle$ consists of the name of a class c with its parameters instantiated with contexts $p_{i\in 1..n}$ that are in scope. Again the first parameter is the owner of objects of that type. If no parameters are required, then the type can be written as just c and the owner is the global context `world`. In the formalism only we sometimes write a type $c\langle\sigma\rangle$, where σ is the binding from context parameters to contexts. The two forms are equivalent: if the formal parameters of c are $\alpha_{i\in 1..n}$, then $\sigma = \{\alpha_i \mapsto p_{i\in 1..n}\}$. This σ can be used as a substitution when, for example, determining the types of fields. We use the notation $\sigma(c\langle p_{i\in 1..n}\rangle)$ to denote $c\langle\sigma(p_i)_{i\in 1..n}\rangle$.

Methods.
Methods are annotated with an effects specification. This indicates the effect shape, or collection of contexts, which may be read and/or written when the method is evaluated. Without loss of generality, methods take only one argument to keep the formal system simple. Examples will use several arguments when necessary.

Expressions.
The expressions encompass the basic object operations: field access, field update, method call, object creation and `null`. Expressions are presented in a form where every intermediate value is stored in a local variable (which cannot be updated). This variable name can subsequently be used to refer to the internal representation context of the value stored in the variable (a feature absent from our earlier system [22]). This enables the tracking of dynamic aliases — references from the stack to objects, rather than from the heap — to the representation of objects. If we were to use a more Java-like syntax, then *final* local variables would be used instead.

Contexts.
Each object ι has an owner which cannot change. The owner is either another object or the context `world`, indicating that it is owned by the system. As we said earlier, the owner o of ι must dominate it, *i.e.,* at every point during

c	$\in$	**ClassName**			
class	$\in$	**Class**	::=	`class` $c\langle\alpha_{i\in 1..n}\rangle$ `extends` $c'\langle p_{i\in 1..n'}\rangle\{fd_{j\in 1..m}\ meth_{k\in 1..p}\}$	
f	$\in$	**FieldName**			
fd	$\in$	**Field**	::=	$t\ f$	
m	$\in$	**MethodName**			
meth	$\in$	**Method**	::=	$t\ m(t\ x)\ \psi\ \{e\}$	
e	$\in$	**Expr**	::=	$z \mid$ `let` $x = b$ `in` e	
x, y	$\in$	**TermVar**			
z	$\in$	**Var**	::=	`this` $\mid x$	
b	$\in$	**Computation**	::=	$z.f \mid z.f = z \mid z.m(z) \mid$ `new` $t \mid$ `null`	
α	$\in$	**ContextVar**			
p, q	$\in$	**Context**	::=	$z \mid \alpha \mid \kappa$	
κ	$\in$	**ActualContext**	::=	`world` $\mid \ldots$	
ϕ	$\in$	**EffectShape**	::=	$\emptyset \mid p.n \mid$ `under`$(p.n) \mid \phi \cup \phi$	n is a natural number
ψ	$\in$	**Effect**	::=	`rd` ϕ `wr` ϕ	
t	$\in$	**Type**	::=	$c\langle p_{i\in 1..n}\rangle$	

Figure 3: Static aspects of Joe$_1$

ι	$\in$	**Location**		
ω	$\in$	**VarOrLoc**	::=	$z \mid \iota$
v	$\in$	**Value**	::=	$\iota \mid$ `null`
κ	$\in$	**ActualContext**	::=	$\ldots \mid \iota$

Figure 4: Some Dynamic Aspects

execution, every path from the root to ι must pass through o. For example, in Figure 2 the `Link` objects are owned by the `List` object, which is in turn owned by the `Main` object, which, along with the `Data` objects, are owned by `World`. It can easily be proven that the ownership relationship forms a tree with `world` as the root [49]. The transitive closure of this relationship is dubbed *inside* and captures the nesting between objects.

There are two forms of context, those which occur in the program source, called just *contexts*, p, and those which occur at run-time, called *actual contexts*, κ, and denote the actual object owners — `world` and one for each object created. Contexts statically refer to actual contexts in the type system. Contexts consist of `world`, which in both program source and at run-time denotes the root context, context parameters (denoted by α_i), and variables z, which denotes the owner of objects owned by the object stored in that variable. In particular, `this` as a context is the owner of the representation. Classes are parameterised by context variables which allows the distinction between lists whose data have different owners. Constraints on contexts statically capture the desired nesting between actual contexts, required to maintain the owners-as-dominators property.

Effect Shapes and Effects. An *effect* consists of a pair of effects shapes, one capturing the contexts which may be read, the other capturing contexts which may be written. Thus `rd` ϕ `wr` ϕ' denotes that a read may occur to a context in ϕ (or ϕ') and that a write may occur to a context within ϕ'. We consider a write effect to include reads which may occur to the same contexts, following Greenhouse and Boyland [30]. That is, the read denoted in the effect above may occur to any context within $\phi \cup \phi'$. This fact is reflected in a single type rule. *i.e.,* in (FXEQ-READINWRITE).

Effects shapes denote collections of contexts. The shape $\emptyset$ represents the empty collection. For every context p and for each $n \geq 0$, there are the shapes $p.n$, referred to as a *band*, and `under`$(p.n)$, referred to as an *under* shape. These shapes are depicted in Figure 5, which shows objects under the tree ordering. The shape $p.n$ is the set of contexts exactly n steps beneath p (in the context tree/ordering). Thus $p.0$ is the singleton set $\{p\}$. When $n = 0$, it is sufficient to write just p or `under`(p). The shape `under`$(p.n)$ corresponds to $\bigcup_{i\geq n} p.i$, and the shape `under(world)` corresponds to the union of all contexts, since `world` is the top context. The union of effect shapes is denoted by $\phi \cup \phi'$.

We have chosen this collection of shapes to enable the abstract representation of effects on unknown parts of an object's internals, while retaining some precision when dealing with these internals. The obvious effect shape, corresponding to `under`(p), denoting some internal object, is not precise enough. Examples in this paper use effects shapes where $n = 0$ or 1. Greenhouse and Boyland [30] offer an alternative but compatible approach.

4. EXAMPLES

Consider the example in Figure 6. It consists of a list and its iterator implemented in our language (after compressing *let* expressions). The list iterator is created as part of the list's representation, and can thus access the rest of the list's representation, namely the links. It can be accessed externally via a dynamic alias. The effect on the `add` method could instead be `wr this` $\cup$ `this.1`, but this is included in `wr under(this)`, the effect we use. Similarly, the effect on the `elem` method could be `wr o.1` $\cup$ `this`, but since `o` is the owner of `this`, the shape `this` (actually `this.0`) is included in `o.1`, and thus the additional information is superfluous.

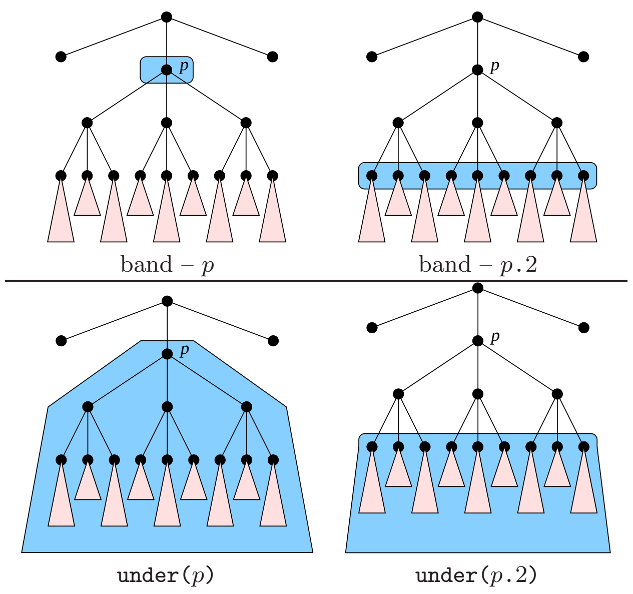

Figure 5: Basic Effects Shapes

```
 class Link<owner,data> {
   Link<owner,data> next;
   Data<data> data;
   Link(Link<owner,data> next, Data<data> data)
          writes this {
     this.next = next; this.data = data;
  }
}

class List<owner,data> {
  Link<this,data> head;
  void add(Data<data> d) writes under(this) {
    head = new Link<this,data>(head, d);
  }
  Iterator<this,data> makeIterator()
        reads this writes this.1 {
    return new Iterator<this,data>(head);
  }
}

class Iterator<o,d> {
  Link<o,d> current;
  Iterator(Link<o,d> first) writes this {
    current = first;
  }
  void next() reads o.1 writes this
    { current = current.next; }
  Data<d> elem() reads o.1
    { return current.data; }
  boolean done() reads this
    { return (current == null); }
}
```

Figure 6: A list and its iterator

A method with encapsulated effects. The first thing of interest in this example is that our system can express that the effects of a method, either read, write, or both, are encapsulated within the object on which the method operates. This is the kind of thing a formal reasoner desires.[1] For example, the effect of method `add` is `wr under(this)`, indicating that it affects only the target object and its representation. Thus, for `list` with type `List<p,world>`, the expression

```
list.add(new Data<world>);
```

has the effect `wr under(list)`, meaning that it only affects the internal representation of `list`. Therefore, the execution of the above expression is guaranteed not to interfere with (*i.e.,* not to modify the fields of) any other object except for `list` and the objects owned by `list`.

Representation access and dynamic aliases. The first incarnation of ownership types forbade dynamic aliases [22]. This limited the expressiveness of the programming language, preventing short-lived abstractions such as a list iterator which must access the representation of another object. The present type system allows such dynamic aliases, while keeping track of the dependencies in a pleasing manner and limiting their lifetimes, without imposing limitations such as restricting references to read-only [44].

For example, here we create a list and an iterator for it:

```
List<p,world> list = new List<p,world>;
Iterator<list,world> iter = list.makeIterator();
```

Notice that the type of the iterator `iter` has the variable name `list` as its owner. This means that `iter` is part of, and is allowed to access, the list's representation, but that `iter` is externally accessible only where `list` is in scope. Once `list` is out of scope, the type `Iterator<list,world>`, and any other type containing context `list` is illegal. Furthermore, since any supertype of `Iterator<list,world>` must mention the context `list`, subtyping does not break this property. Dynamic aliases briefly break the owners-as-dominators property, but only with stack-based or local variables.

Dynamic aliases also permit friendly functions, *i.e.,* functions which can access the representation of different objects, without mixed up the different representations. For example, the following method could be part of the class `List`:

```
bool equals(List<owner,data> other) {
  Link<this,data>  thislink  = this.head;
  Link<other,data> otherlink = other.head;
  ... thislink.data.equal(otherlink.data) ...
}
```

Our type system also allows the initialisation of an object's representation with externally created objects, to which we can guarantee no external aliases — this problem was identified by Detlefs, Leino and Nelson [26]. For example, we require that the Socket underlying the following StockQuoteHandler class be externally created, so that the client can decide on its implementation, and then become a part of the StockQuoteHandler's representation.

[1]Personal communication, Bart Jacobs, November 2001.

```
class StockQuoteHandler {
  Socket<this> s;
  void initSocket(Socket<this> s) writes this {
    this.s = s;
  }
  ...
}
class Main {
  void serveQuotes(...) writes ... {
    StockQuoteHandler h = new StockQuoteHandler();
    Socket<h> s = ...;
    h.initSocket(s);
  }
}
```

We achieve this using the variable holding the StockQuoteHandler, namely `h`, as the owner of the newly created Socket. This context externally denotes the StockQuoteHandler's representation. (The separation between construction and initialisation in this example could be eliminated using a simple static analysis.)

AliasJava [3] uses uniqueness to achieve this behaviour. We gain the encapsulation guarantees which ownership types enable, without requiring uniqueness to transfer the Socket from where it is created to where it is stored. Although we avoid the complications uniqueness precipitates, we cannot however guarantee that the resulting reference to the Socket is unique within the StockQuoteHandler.

While allowing dynamic aliases does weaken the possible encapsulation, it can be further controlled in a full programming language using privacy annotations. For example, we could permit only certain dynamic aliases such as iterators, but prohibit direct access to a list's links. A consequence of deep ownership is that dynamic aliases cannot be stored in an object's field (except one inside the object which owns it). This would be permitted in a system with shallow ownership, which is why the results presented here do not hold in that case. So dynamic aliases represent a sensible compromise which permits certain idioms, such as the ones discussed here, without losing the desired properties following from the owners-as-dominators property.

The present treatment of these examples — initialisation, iterators and friendly functions — is more pleasing than previous attempts [19, 20], because the type of each dynamic alias contains the name of the entity whose representation it dynamically aliases.

5. STATIC SEMANTICS

The static semantics describe well-formed programs. The type rules are defined with respect to some program P, which is left implicit to avoid unnecessary clutter. The program P is also used to populate the dictionaries and helper functions described later.

Most type rules are defined against an environment E which provides types for the free term variables and locations (for the well-formedness of the dynamics) and constraints on context variables:

$$E \quad ::= \quad \emptyset \;\mid\; E, \omega : t \;\mid\; E, \alpha \prec^* p \;\mid\; E, \alpha \succ^* p$$

Although $p \prec^* q$ is equivalent to $q \succ^* p$, we have both $\alpha \prec^* p$ and $\alpha \succ^* p$ in environments, because this allows the newly introduced parameter, in this case denoted by α, to be on the left-hand side of the relation and thus in $\mathrm{dom}(E)$, the collection of free variables.

The dynamic semantics is defined (later) in terms of an explicit binding of free variables, rather than via substitution. This does not affect the static semantics of the programming language, but the bindings are required in the type rules to prove the type preservation theorem. To avoid duplicating work, we include a binding list in the assumption set of most judgements. The bindings B map context variables to actual contexts and variables to values:

$$B \quad ::= \quad \emptyset \;\mid\; B, \alpha = \kappa \;\mid\; B, z = v$$

Note, contexts are defined modulo equality in the binding list (via rules (IN-BIND1) and (IN-BIND2) in Section 8). Thus effect shapes, effects, and types must also be defined modulo context equality. To keep the presentation tractable, however, we have elided the rules dealing with such equalities.

The following judgements define the type system:

$E; B \vdash \diamond$	good environment/binding pair
$E; B \vdash p$	good context
$E; B \vdash p\,\mathsf{R}\,p'$	context p is R-related to p'
	$\mathsf{R} \in \{=, \prec, \prec^+, \prec^*\}$
$E; B \vdash \phi$	good effect shape
$E; B \vdash \phi \sqsubseteq \phi'$	ϕ is a subshape of ϕ'
$E; B \vdash \psi$	good effect
$E; B \vdash \psi \sqsubseteq \psi'$	ψ is a subeffect of ψ'
$E; B \vdash t$	good type
$E; B \vdash t \leq t'$	type t is a subtype of t'
$E; B \vdash v : t$	value v has type t
$E; B \vdash e : t!\psi$	expression e has type t;
	evaluation produces at most effect ψ
$E; B \vdash \mathit{meth}$	good method
$\vdash \mathit{class}$	good class
$\vdash P$	good program

We shall first establish some notation:

$E \vdash \Im$ means $E; \emptyset \vdash \Im$, for each appropriate right-hand side.

$\mathsf{owner}(t)$ takes the owner from a type, where classes without parameters have the manifest owner `world`. $\mathsf{arity}(c)$ is the number of parameters expected by a class.

$$\begin{array}{rcl} \mathsf{owner}(c\langle\rangle) & \hat{=} & \texttt{world} \\ \mathsf{owner}(c\langle\alpha_{i\in 1..n}\rangle) & \hat{=} & \alpha_1, \text{ where } n > 0 \\ \mathsf{arity}(c) & \hat{=} & n, \text{ where } \texttt{class}\ c\langle\alpha_{i\in 1..n}\rangle\{\ldots\} \in P \end{array}$$

$\mathsf{names}(\mathit{meths})$ extracts the method names $\{m_{k\in 1..p}\}$ from $\mathit{meths} = \{t_k\ m_k(t'_k\ x_k)\ldots\{\ldots\}\ _{k\in 1..p}\}$.

$\mathcal{F}$: **ClassName** $\rightarrow_{\mathrm{fin}}$ **FieldMap** is the map from classes to their field list, where **FieldMap** $\hat{=}$ **FieldName** $\rightarrow_{\mathrm{fin}}$ **Type**. We write $\mathcal{F}_c(f)$ instead of $\mathcal{F}(c)(f)$. $\mathcal{F}_c(f)$ gives the type of field f from class c or one of its super classes, and $\mathrm{dom}(\mathcal{F}_c)$ gives the collection of field names.

$\mathcal{MT}$: (**ClassName**$\times$**MethodName**)$\rightarrow_{\mathrm{fin}}$**MethodSpec** is the map from classes to their method interface, where **MethodSpec** $\hat{=}$ **TermVar** $\times$ (**Type** $\times$ **Type** $\times$ **Effect**). $\mathcal{MT}_c(m)$ gives the argument variable and type of method m from class c, denoted $(y, t \rightarrow t'!\psi)$, where y is the argument variable, t is the argument type, t' is the return type, and ψ is the expected effect.

Both $\mathcal{F}$ and $\mathcal{MT}$ take into consideration any parameter instantiations which occur during subclassing. For example,

$\mathcal{F}_{\texttt{List}}(\texttt{head}) = \texttt{Link}\langle\texttt{this},\texttt{data}\rangle$, whereas $\mathcal{F}_{\texttt{DoubleList}}(\texttt{head}) = \texttt{Link}\langle\texttt{this},\texttt{d1}\rangle$, for class `DoubleList` from Figure 7.

$m \uplus m'$ denotes the disjoint union of finite maps m and m', requiring that their domains are disjoint.

Finally, σ_z is the substitution $\sigma \uplus \{\texttt{this} \mapsto z\}$, for any substitution σ. This construction is needed when accessing a field or method through a variable z of type $c\langle\sigma\rangle$, namely, when determining the actual types/effect based on the type of the variable and the type declared in the class. To do this we must also convert the internal name `this`, which can appear in types and effects, to an external name. Therefore, we use the variable name z and extend the substitution σ to σ_z. Thus a type t or effect ψ occurring in class c will become $\sigma_z(t)$ or $\sigma_z(\psi)$ when accessed through variable z. Note that when z is `this`, this mechanism behaves correctly as an internal access. Also, for type $c\langle\rangle$, the empty substitution $\emptyset$ is used, and thus $\emptyset_z$ is the substitution $\{\texttt{this} \mapsto z\}$.

5.1 The Type Rules

We present the type rules as a series of fragments, with a description following each. The rules for well-formed environments have been omitted, but are standard.

Contexts

$$\text{(CTX-VAR)}\ \frac{E;B \vdash \diamond \quad \alpha\,\mathsf{R}\,p \in E}{E;B \vdash \alpha} \qquad \text{(CTX-REP)}\ \frac{E;B \vdash \omega : t}{E;B \vdash \omega} \qquad \text{(CTX-WORLD)}\ \frac{E;B \vdash \diamond}{E;B \vdash \texttt{world}}$$

In rule (CTX-VAR) R is either $\prec^*$ or $\succ^*$. Rule (CTX-REP) permits every variable (and location) to denote a context, ω, which is the representation context of the object referred to by the variable (or location).

Contexts are ordered using relations $\prec$, $\prec^+$ and $\prec^*$, which we refer to as *directly inside*, *strictly inside* and *inside*, respectively. As the notation suggests, $\prec^+$ is the transitive closure of $\prec$, whereas $\prec^*$ is its reflexive, transitive closure. These relations satisfy the following inclusions $\prec\ \subseteq\ \prec^+\ \subseteq\ \prec^*$ and $=\ \subseteq\ \prec^*$. Their composition, $\mathsf{R};\mathsf{R}'$, is defined as follows: $\prec;\prec\ \equiv\ \prec;\prec^*\ \equiv\ \prec^*;\prec\ \equiv\ \prec^+; \mathsf{R} \equiv \mathsf{R};\prec^+\ \equiv\ \prec^+$, and $=;\mathsf{R} \equiv \mathsf{R};= \equiv \mathsf{R}$, and $\prec^*;\prec^*\ \equiv\ \prec^*$. Note that these relations do not appear explicitly in the program text, but are the key to preserving the owners-as-dominators property and for defining our effect shapes.

Context Ordering

$$\text{(IN-ENV)}\ \frac{E;B \vdash \diamond \quad \alpha\,\mathsf{R}\,p \in E}{E;B \vdash \alpha\,\mathsf{R}\,p} \qquad \text{(IN-REFL)}\ \frac{E;B \vdash p}{E;B \vdash p = p}$$

$$\text{(IN-WORLD)}\ \frac{E;B \vdash p}{E;B \vdash p \prec^* \texttt{world}} \qquad \text{(IN-REP)}\ \frac{E;B \vdash \omega : t \quad p = \mathsf{owner}(t)}{E;B \vdash \omega \prec p}$$

$$\text{(IN-WEAKEN)}\ \frac{E;B \vdash p\,\mathsf{R}\,q \quad \mathsf{R} \subseteq \mathsf{R}'}{E;B \vdash p\,\mathsf{R}'\,q} \qquad \text{(IN-TRANS)}\ \frac{E;B \vdash p\,\mathsf{R}\,q \quad E;B \vdash q\,\mathsf{R}'\,q'}{E;B \vdash p\,\mathsf{R};\mathsf{R}'\,q'}$$

The most important rule is (IN-REP) which populates the relations. It states that the representation context is directly inside the owner, as given in a variable's type. The other rules handle constraints on context variables (IN-ENV), relational properties (IN-REFL), (IN-WEAKEN), and (IN-TRANS), and establish that `world` is the greatest element in the partial order (IN-WORLD).

Effect Shapes

$$\text{(FX-}\emptyset\text{)}\ \frac{E;B \vdash \diamond}{E;B \vdash \emptyset} \qquad \text{(FX-BAND)}\ \frac{E;B \vdash p \quad n \geq 0}{E;B \vdash p.n}$$

$$\text{(FX-UNDER)}\ \frac{E;B \vdash p \quad n \geq 0}{E;B \vdash \texttt{under}(p.n)} \qquad \text{(UNION-FX)}\ \frac{E;B \vdash \phi \quad E;B \vdash \phi'}{E;B \vdash \phi \cup \phi'}$$

Each kinds of effect shape is valid for all contexts in scope.

Subshaping

$$\text{(SUBSHAPE-}\emptyset\text{)}\ \frac{E;B \vdash \phi}{E;B \vdash \emptyset \sqsubseteq \phi} \qquad \text{(SUBSHAPE-REFL)}\ \frac{E;B \vdash \phi}{E;B \vdash \phi \sqsubseteq \phi}$$

$$\text{(SUBSHAPE-TRANS)}\ \frac{E;B \vdash \phi \sqsubseteq \phi' \quad E;B \vdash \phi' \sqsubseteq \phi''}{E;B \vdash \phi \sqsubseteq \phi''}$$

$$\text{(SUBSHAPE-UNION)}\ \frac{E;B \vdash \phi \sqsubseteq \phi'' \quad E;B \vdash \phi' \sqsubseteq \phi'''}{E;B \vdash \phi \cup \phi' \sqsubseteq \phi'' \cup \phi'''}$$

$$\text{(SUBSHAPE-DIRECT)}\ \frac{E;B \vdash p \prec q \quad n \geq 0}{E;B \vdash p.n \sqsubseteq q.n{+}1} \qquad \text{(SUBSHAPE-BAND)}\ \frac{E;B \vdash p \quad n \geq 0}{E;B \vdash p.n \sqsubseteq \texttt{under}(p.n)}$$

$$\text{(SUBSHAPE-UNDER)}\ \frac{E;B \vdash p \prec^* q \quad n \geq m}{E;B \vdash \texttt{under}(p.n) \sqsubseteq \texttt{under}(q.m)}$$

The first four rules express simple set theoretic properties of effect shapes. The last three subshaping rules are more interesting. The first, (SUBSHAPE-DIRECT), captures that the contexts n levels below some context p are included within the contexts $n+1$ levels below the context q which is directly above. From rule (SUBSHAPE-BAND), any band context is included in the corresponding under context. Rule (SUBSHAPE-UNDER) lifts the ordering of contexts to under effects, observing that under effects correspond to downward closure, and also captures that the under effect at a particular level contains all deeper under effects.

Effects and Subeffecting

$$\text{(FX)}\ \frac{E;B \vdash \phi_r \quad E;B \vdash \phi_w}{E;B \vdash \texttt{rd}\ \phi_r\ \texttt{wr}\ \phi_w}$$

$$\text{(SUBFX-BASIC)}\ \frac{E;B \vdash \phi_r \sqsubseteq \phi_r' \quad E;B \vdash \phi_w \sqsubseteq \phi_w'}{E;B \vdash \texttt{rd}\ \phi_r\ \texttt{wr}\ \phi_w \sqsubseteq \texttt{rd}\ \phi_r'\ \texttt{wr}\ \phi_w'}$$

$$\text{(FXEQ-READINWRITE)}\ \frac{E;B \vdash \texttt{rd}\ \phi_r\ \texttt{wr}\ \phi_w}{E;B \vdash \texttt{rd}\ \phi_r\ \texttt{wr}\ \phi_w = \texttt{rd}\ \phi_r \cup \phi_w\ \texttt{wr}\ \phi_w}$$

An effect can be formed, by rule (FX), using any valid effect shapes. Rule (SUBFX-BASIC) lifts the subshaping relation to effects. Rule (FXEQ-READINWRITE) captures that a write effect includes reads to the same effect shape.

Types

(TYPE)

$$\frac{\mathsf{arity}(c) = n \quad E;B \vdash p_1 \prec^* p_i \quad \forall i \in 1..n}{E;B \vdash c\langle p_{i\in 1..n}\rangle}$$

Rule (TYPE) enforces that types can be constructed from any class using any context in scope, so long as the correct number of arguments are supplied, and that the owner (first) parameter, if present, is provably inside all other parameters. *This is required ensure that the owners-as-dominators property is maintained.* For example, inside the the body of class `Iterator⟨o,d⟩` the type `Link⟨o,d⟩` is well-formed, but the type `Link⟨d,o⟩` is not well-formed because the owner parameter `d` cannot be proved to be inside `o`. The requirement that the owner parameter be provably inside the remaining context parameters matches the well-formedness rule for classes (CLASS), and ultimately ensures both that the owners-as-dominators property holds and the soundness of effect disjointness tests (see Section 7). More expressive possibilities exist, for example, by allowing the programmer to declare the expected relationship between context parameters to a class (see the first author's dissertation [20]).

Subtyping

(SUB-REFL)

$$\frac{E;B \vdash t}{E;B \vdash t \leq t}$$

(SUB-TRANS)

$$\frac{E;B \vdash t \leq t' \quad E;B \vdash t' \leq t''}{E;B \vdash t \leq t''}$$

(SUB-CLASS)

$$\frac{E;B \vdash c\langle\sigma\rangle \quad \texttt{class } c\langle\alpha_{i\in 1..m}\rangle \texttt{ extends } c'\langle p_{i\in 1..n}\rangle\{\ldots\}}{E;B \vdash c\langle\sigma\rangle \leq c'\langle\sigma(p_i)_{i\in 1..n}\rangle}$$

Subtyping is a reflexive (SUB-REFL) and transitive (SUB-TRANS) relation generated from subclassing (SUB-CLASS), taking into account the instantiation of context parameters. *e.g.,* for `class List⟨owner,data⟩ extends Object⟨owner⟩`, we derive the substitution {`owner` ↦ `p`, `data` ↦ `q`} from type `List⟨p,q⟩`, from which follows `List⟨p,q⟩` ≤ `Object⟨p⟩`.

Subtyping does not permit the owner or any other context parameter to vary, as this would be unsound. This can be seen in the following example:

```
class Test<owner> {
    Test<owner> other;
    void fiddle() writes owner.1 { ... }
}
```

Assume that $E \vdash p \prec q$. Allowing the owner to vary would permit $E \vdash \texttt{Test}\langle p\rangle \leq \texttt{Test}\langle q\rangle$ to be a valid subtyping relation. Assume now that $x : \texttt{Test}\langle p\rangle$, $y : \texttt{Test}\langle q\rangle$, and that x and y are aliases. If the subtyping relation was valid, we would firstly be able to read and write values of type $\texttt{Test}\langle q\rangle$ to and from the `other` field of x, which could be used to break encapsulation. Secondly, the computations $x.\texttt{fiddle}()$ and $y.\texttt{fiddle}()$, would have effects $\texttt{wr}\ p.1$ and $\texttt{wr}\ q.1$ which are distinct in our system, even though x and y refer to the same object. Similar problems would arise, if we considered $E \vdash \texttt{Test}\langle q\rangle \leq \texttt{Test}\langle p\rangle$ to be a valid subtyping.

Values

(VAL-NULL)

$$\frac{E;B \vdash t}{E;B \vdash \texttt{null} : t}$$

(VAL-ω)

$$\frac{\omega : t \in E \quad E;B \vdash \diamond}{E;B \vdash \omega : t}$$

(EXP-FROMVAL)

$$\frac{E;B \vdash v : t}{E;B \vdash v : t!\emptyset}$$

From (VAL-NULL), `null` can have any type. From (VAL-ω), variables (and locations) are given the type which is declared in the environment. The rule (EXP-FROMVAL) allows a value to be treated as an expression whose evaluation causes no effect.

Notation: for the expression rules we write $\emptyset$ to denote no effects, omit either the read or write part when it is empty, and use $\psi\cup\psi'$ to denote union of the underlying components.

Expressions

(EXP-NEW)

$$\frac{E;B \vdash t}{E;B \vdash \texttt{new}\ t : t!\emptyset}$$

(EXP-LET)

$$\frac{\begin{array}{c}E;B \vdash b : t!\psi_1 \quad E, x : t;B \vdash e : t'!\psi_2 \\ E, x : t;B \vdash \psi_1 \cup \psi_2 \sqsubseteq \psi \quad E;B \vdash \psi\end{array}}{E;B \vdash \texttt{let}\ x = b\ \texttt{in}\ e : t'!\psi}$$

(EXP-FIELD)

$$\frac{E;B \vdash z : c\langle\sigma\rangle \quad \mathcal{F}_c(f) = t}{E;B \vdash z.f : \sigma_z(t)!\texttt{rd}\ z}$$

(EXP-UPDATE)

$$\frac{E;B \vdash z : c\langle\sigma\rangle \quad \mathcal{F}_c(f) = t' \quad E;B \vdash y : \sigma_z(t')}{E;B \vdash z.f = y : \sigma_z(t')!\texttt{wr}\ z}$$

(EXP-CALL)

$$\frac{\begin{array}{c}E;B \vdash z : c\langle\sigma\rangle \quad \mathcal{MT}_c(m) = (y', t \to t'!\psi) \\ E;B \vdash y : \sigma_z(t) \quad \sigma' \equiv \sigma \uplus \{y' \mapsto y\}\end{array}}{E;B \vdash z.m(y) : \sigma'_z(t')!\sigma'_z(\psi)}$$

(EXP-SUB)

$$\frac{E;B \vdash e : t!\psi \quad E;B \vdash t \leq t' \quad E;B \vdash \psi \sqsubseteq \psi'}{E;B \vdash e : t'!\psi'}$$

Object creation, rule (EXP-NEW), is valid for any type in scope and produces no effect. The (EXP-LET) rule follows the usual pattern for *let* expressions, where the effects of each part of the expression are collected. Furthermore, the two additional clauses ensure that the resulting effect does not involve the local variable x, as this would mean that x would be no longer in scope. The type of a field, rule (EXP-FIELD), is as declared in the appropriate class, modulo the instantiation of context parameters given by substitution σ. To handle the internal variable `this`, the substitution is extended with the mapping to give the local variable name z to `this`. In addition, a read effect is produced on context z. The rule (EXP-UPDATE) employs the same trick used in (EXP-FIELD), though it also requires the value being placed in the field to have the appropriate type. In addition, updating a field produces a write effect on the context z. For method call, rule (EXP-CALL) produces the effects latent in the method, as declared in the class. The extended substitution σ'_z is applied to the effect expression, which may refer to `this` or the method argument, to convert it in terms of the names in scope. Subtyping/subeffecting (EXP-SUB) follows the usual pattern [46].

If variable x has type $\texttt{List}\langle a, b\rangle$, then $\sigma_x = \{\texttt{owner} \mapsto a, \texttt{data} \mapsto b, \texttt{this} \mapsto x\}$. The type of the `head` field of class `List` is declared as $\texttt{Link}\langle\texttt{this}, \texttt{data}\rangle$. Thus x`.head` has type $\sigma_x(\texttt{Link}\langle\texttt{this}, \texttt{data}\rangle) = \texttt{Link}\langle x, b\rangle$. Pleasingly, x remains in the type, tracking the owner of this representation, namely the object in variable x.

Now let's discuss briefly scoping and the rule (EXP-LET). Consider just the types for now. The type of the expression

$$\texttt{let}\ x = \texttt{new List}\langle a, b\rangle\ \texttt{in let}\ z = x.\texttt{head in}\ z$$

is *not* $\texttt{Link}\langle x, b\rangle$, as we might expect, because this type contains x which is no longer in scope. Indeed this expression cannot be typed, because there is no supertype which does not include x, since the owner cannot be forgotten. Thus the representation cannot escape beyond the scope of the object that owns it. It is only a short-lived, dynamic alias.

But we do require that effects escape beyond the scope of variables which denote them, since the effects have a more global consequence. Consider

$$\begin{array}{l}\texttt{let}\ x = \texttt{new List}\langle a, b\rangle\ \texttt{in let}\ z = x.\texttt{head in} \\ \qquad\quad \texttt{let}\ y = \texttt{null in}\ y\end{array}$$

The only subexpression which produces an effect is the computation x`.head`. It produces effect `rd` x. But x is not in scope at the end of the expression, so we must use subeffecting to find a suitable supereffect which includes this effect. One such effect is `rd` a`.1`.

Note, that the effect `rd` a`.1` is a safe, but imprecise overestimation, since this expression only reads the field of a locally created object. In general, with appropriate effects encapsulation, we ought to be able to report *no* effect for certain expressions, but this is left for future work.

Method

(METHOD)

$$\frac{E, y : t \vdash e : t'!\psi}{E \vdash t'\ m(t\ y)\ \psi\ \{e\}}$$

In rule (METHOD) the premise ensures that the method body has the type and effect which are declared — propagated downwards from rule (CLASSES). This may include shapes involving the argument y. This enables effects (and the return type) to be defined in terms of the argument variable, permitting the statement that the argument is read or written.

Classes

(CLASS-OBJECT)

$$\frac{}{\vdash \texttt{class Object}\langle\alpha_1\rangle\{\ \}}$$

(CLASS)

$$\frac{\begin{array}{ll}
(1) & E \equiv \alpha_1 \prec^* \texttt{world}, (\alpha_i \succ^* \alpha_1)_{i\in 2..n}, \texttt{this} : c\langle\alpha_{i\in 1..n}\rangle \\
(2) & E \vdash c'\langle\sigma\rangle \qquad \mathsf{owner}(c\langle\alpha_{i\in 1..n}\rangle) = \mathsf{owner}(c'\langle\sigma\rangle) \\
(3) & \{f_{i\in 1..m}\} \cap \mathrm{dom}(\mathcal{F}_{c'}) = \emptyset \qquad E \vdash t_{j\in 1..m} \\
(4) & E \vdash meth_{k\in 1..p} \\
(5) & \begin{array}{l}\forall m \in \mathsf{names}(meth_{k\in 1..p}) \\ \quad \cap\ \mathrm{dom}(\mathcal{MT}_{c'})\end{array} \left\{\begin{array}{ll}\mathcal{MT}_c(m) \equiv t \rightarrow t'!\psi & \text{(a)} \\ \mathcal{MT}_{c'}(m) \equiv t'' \rightarrow t'''!\psi' & \text{(b)} \\ t \equiv \sigma(t'') \quad t' \equiv \sigma(t''') & \text{(c)} \\ E; \emptyset \vdash \psi \sqsubseteq \sigma(\psi') & \text{(d)}\end{array}\right.
\end{array}}{\vdash \texttt{class}\ c\langle\alpha_{i\in 1..n}\rangle\ \texttt{extends}\ c'\langle\sigma\rangle\{t_j\ f_{j\in 1..m}\quad meth_{k\in 1..p}\}}$$

Sitting at the top of the hierarchy is the class `Object` which has a single owner parameter.

Aspects of the rule (CLASS) have been numbered. Firstly (1), a class is checked against an environment consisting of an unbounded owner parameter α_1 and a number of other parameters which are outside ($\succ^*$) of it. *This is the minimal requirement for ensuring the owners-as-dominators property and soundness* [20]. In addition, `this` is present at the type being defined. Secondly (2), the `extends` clause is valid when the type $c'\langle\sigma\rangle$ is valid, requiring that the constraints c' imposes are satisfied, and that ownership is preserved. Thirdly (3), the fields declared in this class must be distinct from the inherited fields, and their types must be well-formed. Next (4) checks that the methods declared in the current class are okay. Finally (5), for each overridden method: (a) the method type declared in the current class and (b) the method type declared for the superclass (naturally taking any prior subclassing into account), (c) must have the same argument and return types modulo the instantiation of the superclass's parameters, and (d) the effect declared on the current method is a subeffect of the effect on the corresponding method in the superclass, modulo the instantiation of parameters.

Some of these issues are illustrated in Figure 7, where the class $\texttt{DoubleList}\langle o, d1, d2\rangle$ extends the class $\texttt{List}\langle o, d\rangle$. The associated environment for checking the class body is $E \equiv o \prec^* \texttt{world}, d1 \succ^* o, d2 \succ^* o, \texttt{this}{:}\texttt{DoubleList}\langle o, d1, d2\rangle$ and associated substitution is $\sigma \equiv \texttt{owner} \mapsto o, \texttt{data} \mapsto d$. The method `f` is redefined in `DoubleList`. The return type of `f` in class `DoubleList` is $\texttt{Data}\langle\texttt{data}\rangle$, while in class `DoubleList` it is is $\texttt{Data}\langle d1\rangle$ – thus, requirements (5.a-c) are satisfied. Finally, the effect of function `f` in `DoubleList` is a subeffect of the effect of `f` in `List`, *i.e.,*

$$E \vdash \texttt{wr under(this)} \sqsubseteq \sigma(\texttt{wr under(owner)}).$$

Therefore requirement (5.d) is satisfied.

On the other hand, a class $\texttt{OtherList}\langle o, d1, d2\rangle$ extending the class $\texttt{List}\langle d1, o\rangle$ would be illegal, as $d1$ cannot be proven to be inside o in the environment E, and so requirement (2) would not be satisfied. Also, if `f` in class `DoubleList` had the return type $\texttt{Data}\langle d2\rangle$, then requirement (5.c) would be violated. Lastly, if `f` in class `DoubleList` had the effect `wr under(d2)`, then requirement (5.d) would be violated.

Programs

(PROG)

$$\frac{\vdash class_j \quad \forall class_j \in P}{\vdash P}$$

```
 class List<owner,data> {
  Link<this,data> head;
  void add(Data<data> d) writes under(this) {
    head = new Link<this,data>(d, head);
  }
  Data<data> f() writes under(owner) {
    ...
  }
}
class DoubleList<o,d1,d2> extends List<o,d1> {
  List<this,d2> otherlist;
  void add2(Data<d2> d) writes under(this) {
     otherlist.add(d);
  }
  Data<d1> f() writes under(this) {
    ...
  }
}
```

Figure 7: Subclasses example code

Finally, a program is well-formed if all its constituent classes are well-formed.

5.2 Properties

An important property satisfied by all judgements, including those presented later, is the Extension Lemma. It states that all judgements are preserved when we add typings for new variables or introduce new parameters into the environment ($E \gg E'$), or when we add new bindings for contexts or variables ($B \gg B'$). That is, any assertion our type system makes remains valid regardless of what configuration it appears in, so long as the assumptions about free variables are satisfied. This includes any possible configurations which may appear during evaluation.

Before giving the lemma we define what it means to extend an environment or binding.

Definition 5.1. *Say that E'* extends *E, denoted $E' \gg E$, whenever E is a subsequence of E'. Similarly define $B' \gg B$.*

Lemma 5.2 (Extension). *Assume that $E; B \vdash \Im$, for some right-hand form $\Im$.*

- *If $E' \gg E$ and $E'; B \vdash \diamond$, then $E'; B \vdash \Im$.*
- *If $B' \gg B$ and $E; B' \vdash \diamond$, then $E; B' \vdash \Im$.*

The first clause is also applicable to judgements which do not have B in the assumption set.

The first part of this lemma corresponds to the usual extension lemma, the second part to the usual substitution lemma found in the meta-theory of many type systems [1].

6. DYNAMIC SEMANTICS

We specify the dynamic behaviour of $\texttt{Joe}_1$ as a large step operational semantics. This semantics uses a heap mapping locations to objects, and bindings of variables to values (just like a stack).

The additional features we introduce for the dynamics of $\texttt{Joe}_1$ are given in Figure 8. Locations ι are the addresses of objects. Values consist of locations and `null`. Since objects are the real owners, we also extend the actual contexts κ with locations. Heaps H are maps from locations to objects. Objects map field names to values, and carry their type. Note, that in contrast with the usual semantics, objects carry their type, *i.e.,* a $c\langle\sigma\rangle$, rather than just a class, c. The extra information in the substitution σ represents the object's owner, and the owners of further objects reachable through the object's fields. The ownership information is not directly needed for evaluation, but it is, along with the bindings B, required for the proof of type preservation, as has also been demonstrated in other settings [48, 42].

The heap from Figure 2 could be represented as

$$H_0 = \left\{ \begin{array}{lcl} \iota_1 & \mapsto & [\texttt{list} \mapsto \iota_4]^{\texttt{Main}\langle\texttt{world}\rangle} \\ \iota_2 & \mapsto & [\ldots]^{\texttt{Data}\langle\texttt{world}\rangle} \\ \iota_3 & \mapsto & [\ldots]^{\texttt{Data}\langle\texttt{world}\rangle} \\ \iota_4 & \mapsto & [\texttt{head} \mapsto \iota_5]^{\texttt{List}\langle\iota_1,\texttt{world}\rangle} \\ \iota_5 & \mapsto & [\texttt{next} \mapsto \iota_6, \texttt{data} \mapsto \iota_2]^{\texttt{Link}\langle\iota_4,\texttt{world}\rangle} \\ \iota_6 & \mapsto & [\texttt{next} \mapsto \texttt{null}, \texttt{data} \mapsto \iota_3]^{\texttt{Link}\langle\iota_4,\texttt{world}\rangle} \end{array} \right\},$$

where ι_1 represents the `Main` object, ι_2 and ι_3 represent the `Data` objects, ι_4 together with ι_5 and ι_6 represent the List with its two Links.

We use the notation $H(\iota).f$ to denote $H(\iota)(f)$, the double lookup of the location ι in map H and field f in the object it returns. For example, $H_0(\iota_5).\texttt{next} = \iota_6$. If o is a finite map, then $o[f \mapsto v]$ denotes the update with binding $f \mapsto v$.

The semantics takes the form of two relations: one to capture basic computations ($\Longrightarrow$), the other for expression evaluation ($\longrightarrow$). The computation relation takes a computation configuration to a final configuration. The evaluation relation takes an expression configuration to a final configuration.

The function $\mathcal{MB}_c(m)$ gives the body of method m from class c as the triple (y, e, σ) consisting of the method's argument y, the expression e constituting the method body, and a map σ from the context parameters of the class where method m is declared to the contexts visible in class c. σ is required to provide the correct actual contexts for the contexts in scope in during the evaluation of the method body e, which in turn is required to provide enough information when proving soundness.

For example, if we want to call the method `add` for an object of class `DoubleList`, then we shall find the method body in class `List` and will need to convert the context parameters of class `List` into context parameters of class `DoubleList`. Thus, we look up though $\mathcal{MB}_{DoubleList}(\texttt{add})$, giving $\{\texttt{owner} \mapsto o, \texttt{data} \mapsto d1\}$ as its third component. On the other hand, if we want to call `add` for an object of class `List`, then we look up through $\mathcal{MB}_{List}(\texttt{add})$, whose third component is equal to $\{\texttt{owner} \mapsto \texttt{owner}, \texttt{data} \mapsto \texttt{data}\}$.

6.1 Evaluation

Evaluation is effectively the same as Java's. The ownership information has no effect on computation. It is threaded through the semantics, as we have said, to facilitate the proof of soundness. In addition to the ownership information, computational effects are recorded.

Computation Rules

(comp-null)

$$\frac{}{\langle H; B; \texttt{null}\rangle \overset{\emptyset}{\Longrightarrow} \langle H; \texttt{null}\rangle}$$

$$\begin{array}{rcllll}
\iota & \in & \textbf{Location} & & \\
v & \in & \textbf{Value} & ::= & \iota \mid \texttt{null} \\
\kappa & \in & \textbf{ActualContext} & ::= & \ldots \mid \iota \\
o & \in & \textbf{Object} & = & \textbf{Type} \times (\textbf{FieldName} \rightarrow_{\text{fin}} \textbf{Value}) \\
 & & & ::= & [f \mapsto v_{f \in \text{dom}(\mathcal{F}_c)}]^{c\langle\sigma\rangle} \\
H & \in & \textbf{Heap} & = & \textbf{Location} \rightarrow_{\text{fin}} \textbf{Object} \\
\langle H; B; w\rangle & \in & \textbf{Config}[W] & = & \textbf{Heap} \times \textbf{Bindings} \times W, \text{ where } w ::= e \mid b \\
\langle H; v\rangle & \in & \textbf{FinalConfig} & = & \textbf{Heap} \times \textbf{Value} \\
\langle H; B; b\rangle \overset{\psi}{\Longrightarrow} \langle H'; v\rangle & \in & \textbf{ComputationRelation} & = & \textbf{Config}[\textbf{Computation}] \times \textbf{Effect} \times \textbf{FinalConfig} \\
\langle H; B; e\rangle \overset{\psi}{\longrightarrow} \langle H'; v\rangle & \in & \textbf{EvaluationRelation} & = & \textbf{Config}[\textbf{Expr}] \times \textbf{Effect} \times \textbf{FinalConfig} \\
 & & \mathcal{MB} & : & (\textbf{ClassName} \times \textbf{MethodName}) \rightarrow_{\text{fin}} \textbf{MethodBody} \\
 & & \textbf{MethodBody} & \widehat{=} & \textbf{Var} \times \textbf{Expr} \times \textbf{CMap} \\
 & & \textbf{CMap} & \widehat{=} & \textbf{ContextVar} \rightarrow_{\text{fin}} \textbf{Context}
\end{array}$$

Figure 8: Dynamic aspects of Joe$_1$

(COMP-NEW)
$$\frac{H' \equiv H \uplus \iota \mapsto [f \mapsto \texttt{null}_{f \in \text{dom}(\mathcal{F}_c)}]^{c\langle B(p_i)_{i \in 1..n}\rangle}}{\langle H; B; \texttt{new}\ c\langle p_{i \in 1..n}\rangle\rangle \overset{\emptyset}{\Longrightarrow} \langle H'; \iota\rangle}$$

(COMP-FIELD)
$$\frac{B(y) = \iota \quad H(\iota).f = v}{\langle H; B; y.f\rangle \overset{\texttt{rd}\ \iota}{\Longrightarrow} \langle H; v\rangle}$$

(COMP-UPDATE)
$$\frac{B(y) = \iota \quad B(y') = v \quad H \equiv H' \uplus \iota \mapsto o}{\langle H; B; y.f = y'\rangle \overset{\texttt{wr}\ \iota}{\Longrightarrow} \langle H' \uplus \iota \mapsto o[f \mapsto v]; v\rangle}$$

(COMP-CALL)
$$\frac{\begin{array}{c} B(y) = \iota \quad H(\iota) = [f \mapsto v_{f \in \text{dom}(\mathcal{F}_c)}]^{c\langle\sigma\rangle} \\ \mathcal{MB}_c(m) = (x, e, \{\alpha_i \mapsto p_{i \in 1..n}\}) \\ \langle H; \alpha_i = \sigma(p_i)_{i \in 1..n}, \texttt{this} = \iota, x = B(y'); e\rangle \overset{\psi}{\longrightarrow} \langle H'; v\rangle \end{array}}{\langle H; B; y.m(y')\rangle \overset{\psi}{\Longrightarrow} \langle H'; v\rangle}$$

By (COMP-NULL), `null` evaluates to itself. By (COMP-NEW), object creation adds a new location to the heap bound to an object of the appropriate type with its fields set to `null`. The new object's exact type (the superscript) is determined by looking up the context variables in the binding list B. The freshness of the new location ι is implicit in the definition of $\uplus$. Each of the other computation rules first looks up each variable in the bindings. Field access, (COMP-FIELD), simply returns the value in the appropriate field of the object and produces an effect indicating which object was read. Similarly, field update, (COMP-UPDATE), writes the value to appropriate field and produces an effect indicating which object was written. The most complicated rule (COMP-CALL) is for method call. Firstly the target object is extracted from the binding list, and, once its type c has been determined, the method lookup is performed. The method body is then evaluated against a new binding which has an actual context for each context parameter of the class in which the method body is defined; `this` bound to the location of the target object; and the method's formal parameter bound to the value bound to the actual parameter.

For example, with binding B_0, containing $\texttt{this} = \iota_4, \texttt{owner} = \iota_1, \texttt{data} = \texttt{world}, \texttt{z} = \iota_3$, and H_0 defined above, the creation of an object of type $\texttt{Link}\langle\texttt{this},\texttt{data}\rangle$ evaluates as

$$\langle H_0; B_0; \texttt{new Link}\langle\texttt{this},\texttt{data}\rangle\rangle \overset{\emptyset}{\Longrightarrow} \langle H_1; \iota_7\rangle$$

with $H_1 = H_0 \uplus \iota_7 \mapsto [\texttt{next} \mapsto \texttt{null}, \texttt{data} \mapsto \texttt{null}]^{\texttt{Link}\langle\iota_4,\texttt{world}\rangle}$. Calling the method `add` for an object of class `List` creates a new `Link`, *i.e.,*

$$\langle H_0; B_0; \texttt{this.add(z)}\rangle \overset{\psi}{\Longrightarrow} \langle H_2; \iota_7\rangle,$$

with

$$\begin{array}{rcl} H_2 & = & H_0[\iota_4 \mapsto [\texttt{head} \mapsto \iota_7]^{\texttt{List}\langle\iota_1,\texttt{world}\rangle}] \\ & \uplus & \iota_7 \mapsto [\texttt{next} \mapsto \iota_5, \texttt{data} \mapsto \iota_3]^{\texttt{Link}\langle\iota_4,\texttt{world}\rangle}. \end{array}$$

Note that we applied the usual semantics for constructors, *i.e.,* object creation followed by field initialisation.

Evaluation Rules

(EV-VAR)
$$\frac{}{\langle H; B; z\rangle \overset{\emptyset}{\longrightarrow} \langle H; B(z)\rangle}$$

(EV-LET)
$$\frac{\langle H; B; b\rangle \overset{\psi}{\Longrightarrow} \langle H'; v\rangle \quad \langle H'; B, x = v; e\rangle \overset{\psi'}{\longrightarrow} \langle H''; v'\rangle}{\langle H; B; \texttt{let}\ x = b\ \texttt{in}\ e\rangle \overset{\psi \cup \psi'}{\longrightarrow} \langle H''; v'\rangle}$$

Variables require no evaluation (EV-VAR), simply a lookup in the binding list. Let expressions, (EV-LET) are evaluated by first performing the computation, binding the resulting value v to the variable x and adding this to the binding list, and then evaluating the other expression with the new heap and binding list.

We have omitted rules for trapping errors such as accessing `null` and for field accesses and updates and method calls on objects which do not have the appropriate field or method, and rules for error propagation.

6.2 Properties of Evaluation

From rule (COMP-UPDATE) it is clear that the write effect produced by evaluation denotes the set of locations which refer to the objects that were written. Thus we can easily show the following lemma which states that objects which are not reported as being written remain unchanged during evaluation.

LEMMA 6.1. *Whenever* $\langle H;\ B;\ e\rangle \overset{rd\ \phi\ wr\ \phi'}{\longrightarrow} \langle H';\ v\rangle$, *or* $\langle H; B; b\rangle \overset{rd\ \phi\ wr\ \phi'}{\Longrightarrow} \langle H'; v\rangle$, *then* $H\mid_{dom(H)\setminus\phi'} = H'\mid_{dom(H)\setminus\phi'}$.

6.3 Typing

To enable the formulation of a type preservation theorem, we need to specify the well-formedness of the dynamic features. First we provide the type rules for bindings, before giving additional judgements for the other aspects of the dynamics.

Bindings

$$\textsc{(binding-context)}\quad \frac{\alpha\,\mathsf{R}\,p \in E \quad E; B \vdash \kappa\,\mathsf{R}\,p \quad \alpha \notin \mathrm{dom}(B)}{E; B, \alpha = \kappa \vdash \diamond}$$

$$\textsc{(binding-value)}\quad \frac{z : t \in E \quad E; B \vdash v : t \quad z \notin \mathrm{dom}(B)}{E; B, z = v \vdash \diamond}$$

$$\textsc{(in-bind1)}\quad \frac{E; B \vdash \diamond \quad \alpha = \kappa \in B}{E; B \vdash \alpha = \kappa} \qquad \textsc{(in-bind2)}\quad \frac{E; B \vdash \diamond \quad z = \iota \in B}{E; B \vdash z = \iota}$$

Bindings can only be added if the binding is not already present and if the assumptions about the actual context (BINDING-CONTEXT) or value (BINDING-VALUE) are satisfied. Both rules (IN-BIND1) and (IN-BIND2) introduce equivalences between contexts. Note that rule (IN-BIND2) does not apply when the binding is $z = \texttt{null}$.

The additional judgements are:

$E \vdash \iota \mapsto o$	good location to object binding
$E \vdash H$	good heap
$E \vdash \langle H; B; w\rangle : t!\psi$	configuration type t produces effect ψ

We write $E \vdash \langle H; B; w\rangle : t$ when the effect is not important, as with result configurations.

Heap Typing

$$\textsc{(object)}\quad \frac{o \equiv [f \mapsto v_{f \in \mathrm{dom}(\mathcal{F}_c)}]^{c\langle\sigma\rangle} \quad E; \emptyset \vdash v_f : \sigma_\iota(\mathcal{F}_c(f)) \quad \forall f \in \mathrm{dom}(\mathcal{F}_c)}{E \vdash \iota \mapsto o : c\langle\sigma\rangle}$$

$$\textsc{(heap)}\quad \frac{\iota : t \in E \quad E \vdash \iota \mapsto o : t \quad \forall \iota \mapsto o \in H}{E \vdash H}$$

By rule (OBJECT) a heap binding is well-formed if the values in the object's fields are well-typed for all the fields in the object's class. The rule (HEAP) states that a heap is well-formed if all the heap bindings are well-formed.

Note, that in (OBJECT) we require each field's value to have the type which results from the substitution σ_ι, where σ is the substitution in the object's type, and ι is the object itself; ι is required when checking the context `this`. Thus, for an environment E containing $\iota_5 : \texttt{Link}\langle\iota_4, \iota_7\rangle$, indicating that the owner of ι_5 is ι_4, we have that

$$E \vdash \iota_4 \mapsto [\texttt{head} \mapsto \iota_5]^{\texttt{List}\langle\iota_1,\iota_7\rangle} : \texttt{List}\langle\iota_1, \iota_7\rangle$$

holds. But:

$$E \not\vdash \iota_4 \mapsto [\texttt{head} \mapsto \iota_5]^{\texttt{List}\langle\iota_1,\iota_8\rangle} : \texttt{List}\langle\iota_1, \iota_8\rangle \text{ and}$$
$$E \not\vdash \iota_9 \mapsto [\texttt{head} \mapsto \iota_5]^{\texttt{List}\langle\iota_1,\iota_7\rangle} : \texttt{List}\langle\iota_1, \iota_7\rangle.$$

Configuration Typing

$$\textsc{(config)}\quad \frac{E \vdash H \quad E; B \vdash w : t!\psi}{E \vdash \langle H; B; w\rangle : t!\psi}$$

We use a single rule to type the different kinds of configurations. The rule (CONFIG) extracts the binding list B from the configuration and uses it to type the expression. These bindings can be used to equate variables which may appear in a type or effect with the object to which they correspond. Thus, if $x = \iota$ appears in the binding, we can prove $E; B \vdash \texttt{wr}\ x = \texttt{wr}\ \iota$ (using obvious equality rules which have been omitted). The effect $\texttt{wr}\ x$ could be the one which the type system anticipates, where as $\texttt{wr}\ \iota$ could be the actual effect which occurs. Being able to exploit the above equivalence is essential for proving type preservation.

The Type Preservation theorem states that evaluation preserves the type given to an expression or computation. Furthermore, any effects which evaluation actually produces (ψ') are included within the predicted effect (ψ).

THEOREM 6.2 (PRESERVATION). *If* $E \vdash \langle H; B; b\rangle : t!\psi$ *and* $\langle H; B; b\rangle \overset{\psi'}{\Longrightarrow} \langle H'; v\rangle$, *then there exists an* E' *such that* $E' \gg E$, $E' \vdash \langle H'; B; v\rangle : t$ *and* $E'; B \vdash \psi' \sqsubseteq \psi$.
Similarly for $\langle H; B; e\rangle \overset{\psi'}{\longrightarrow} \langle H'; v\rangle$.

6.4 Soundness of subeffecting

We now explore what is means for an effect to be included within another effect. To demonstrate the soundness of the subeffecting rules, specifically the subshaping rules, we provide an interpretation of each effect shape into the basic entities (ultimately locations) which it denotes. We do this by generating the basic shape which underlies an effect shape.

DEFINITION 6.3 (BASIC). *An effect shape is* basic *if it consists of a possibly empty union of shapes of the form* $p\,.\,0$.

We can treat basic effect shapes as sets. The following states that for basic effects subshaping behaves like subset:

LEMMA 6.4. *If* $E; B \vdash \phi \sqsubseteq \phi'$, *where* ϕ *and* ϕ' *are basic, then, when considered as sets,* $\phi \subseteq \phi'$.

The function $\phi\downarrow_{E;B}$ generates a basic effect shape which underlies the shape ϕ, for a given environment E and binding B. This can be considered as the set of variables and locations to which the shape corresponds.

$$\begin{aligned}
\emptyset\!\downarrow_{E;B} &\;\widehat{=}\; \emptyset\\
p\,.\,0\!\downarrow_{E;B} &\;\widehat{=}\; \{q \mid E;B \vdash p = q\}\\
(p\,.\,n{+}1)\!\downarrow_{E;B} &\;\widehat{=}\; \{q' \mid E;B \vdash q' \prec q \;\wedge q \in p\,.\,n\!\downarrow_{E;B}\}\\
\texttt{under}(p\,.\,n)\!\downarrow_{E;B} &\;\widehat{=}\; \{q' \mid E;B \vdash q' \prec^* q \wedge q \in p\,.\,n\!\downarrow_{E;B}\}\\
(\phi \cup \phi')\!\downarrow_{E;B} &\;\widehat{=}\; \phi\!\downarrow_{E;B} \cup \phi'\!\downarrow_{E;B}
\end{aligned}$$

By the next lemma, not only is the underlying shape well-formed, but also, it corresponds to precisely the collection of contexts underlying an effect shape (for the given E and B):

Lemma 6.5. *The following hold:*

- *If $E;B \vdash \phi$, then $E;B \vdash \phi\!\downarrow_{E;B}$.*
- *$E;B \vdash p\,.\,0 \sqsubseteq \phi$ if and only if $p \in \phi\!\downarrow_{E;B}$.*

Our last theorem amounts to saying that subeffecting is sound.

Theorem 6.6. *If $E;B\vdash\phi \sqsubseteq \phi'$, then $E;B\vdash\phi\!\downarrow_{E;B} \sqsubseteq \phi'\!\downarrow_{E;B}$.*

An important implication of the above theorem and the preservation lemma is that the actual effects of evaluating an expression are covered by the effects reported by the type system:

Corollary 6.7. *If $E \vdash \langle H;B;b\rangle : t!\texttt{rd}\,\phi_r\;\texttt{wr}\,\phi_w$ and $\langle H;B;b\rangle \overset{\texttt{rd}\,\phi'_r\;\texttt{wr}\,\phi'_w}{\Longrightarrow} \langle H';v\rangle$, then there exists an E', with $E' \gg E$, such that $E' \vdash \langle H';B;v\rangle : t$ and*

- *if $\iota \in \phi'_r$, then $\iota \in \phi_r\!\downarrow_{E';B}$, and*
- *if $\iota \in \phi'_w$, then $\iota \in \phi_w\!\downarrow_{E';B}$.*

Similarly for $\langle H;B;e\rangle \xrightarrow{\psi'} \langle H';v\rangle$.

We could now easily formulate the owners-as-dominators property for our system. Due to space limitations we refer readers to prior work dealing with that [49, 22, 21, 20].

7. DISJOINT TYPES AND EFFECTS

Among the main contributions of this paper are the tests for disjointness of type and effect. These judgements allow us to deduce facts about the aliasing and effects in a program using the rules given here, rather than by relying on a more weighty general theorem prover.

When two types are disjoint, then any two variables or fields having these types cannot be aliases. The disjointness of effect shapes can then be used to determine whether two expressions potentially interfere, as we show in Section 7.2. Both disjointness tests depend upon a simpler test which determines whether two contexts are disjoint (which in turn may depend on the disjointness of types).

The disjointness information stems, ultimately, from a number of facts: that the representation context of an object is different from externally visible contexts, as in rule (in-rep); that some types are non-overlapping due to their relative places in the inheritance hierarchy; and that the nesting which follows from the encapsulation of objects induces a tree-shaped partial order on objects whose structure can be exploited.

The disjointness rules are defined by the additional judgements:

$E;B \vdash p \,\#\, q$	context p is disjoint from q
$E;B \vdash t \,\#\, t'$	type t is disjoint from t'
$E;B \vdash \phi \,\#\, \phi'$	shape ϕ is disjoint from ϕ'
$E;B \vdash \psi \,\#\, \psi'$	non-interference of effects ψ and ψ'

These rules are designed to remain valid for all valid bindings of free variables, that is, for all valid B. For example, the following is invalid

$$\alpha \prec^* \texttt{world}, \beta \prec^* \texttt{world}; \emptyset \vdash \alpha \,\#\, \beta,$$

since the binding $\alpha = \texttt{world}, \beta = \texttt{world}$ would result in

$$\alpha{\prec^*}\texttt{world}, \beta{\prec^*}\texttt{world}; \alpha{=}\texttt{world}, \beta{=}\texttt{world} \vdash \texttt{world} \,\#\, \texttt{world},$$

which is clearly false.

The disjointness relations are symmetric, *i.e.,* we have both $E;B \vdash \Im \,\#\, \Im'$ and $E;B \vdash \Im' \,\#\, \Im$. For space reasons we have omitted the corresponding rules.

Disjointness of Context.

The context disjointness rules guarantees that the contexts are always disjoint, regardless of the bindings to the context parameters. This is stronger than the similar notion of *role separation* in Flexible alias protection [47].

Disjointness of Context

(dctx-neq)
$$\frac{E;B \vdash p \prec^+ p'}{E;B \vdash p \,\#\, p'}$$

(dctx-type)
$$\frac{E;B \vdash \omega : t \quad E;B \vdash \omega' : t' \quad E;B \vdash t \,\#\, t'}{E;B \vdash \omega \,\#\, \omega'}$$

(dctx-loc)
$$\frac{E;B \vdash \diamond \quad \iota, \iota' \in \mathrm{dom}(E) \quad \iota \neq \iota'}{E;B \vdash \iota \,\#\, \iota'}$$

By rule (dctx-neq) context disjointness is first derived from the $\prec^+$ relation, which informs when two contexts, although related, are not equal. We utilise type disjointness in rule (dctx-type) to give that the representation contexts of two variables (or locations) which cannot be aliases are disjoint. Finally, two distinct locations have disjoint representation contexts, from (dctx-loc).

Disjointness of Type.

Our rules for the disjointness of type rely only on the disjointness of class based on the relative positions in the inheritance hierarchy, and on the disjointness of context parameter bindings.

Disjointness of Type

(dtype-class)
$$\frac{E;B \vdash c\langle\sigma\rangle \quad E;B \vdash c'\langle\sigma'\rangle \quad \neg(c\;\texttt{extends}^* c' \vee c'\;\texttt{extends}^* c)}{E;B \vdash c\langle\sigma\rangle \,\#\, c'\langle\sigma'\rangle}$$

(dtype-ctx)
$$\frac{\begin{array}{c}E;B \vdash c\langle p_{i\in 1..n}\rangle \quad E;B \vdash c\langle q_{i\in 1..n}\rangle\\ E;B \vdash p_i \,\#\, q_i \quad \text{for some } i \in 1..n\end{array}}{E;B \vdash c\langle p_{i\in 1..n}\rangle \,\#\, c\langle q_{i\in 1..n}\rangle}$$

(dtype-sub)
$$\frac{E;B \vdash t \,\#\, t' \quad E;B \vdash t'' \leq t'}{E;B \vdash t \,\#\, t''}$$

From rule (DTYPE-CLASS), two types are definitely disjoint if neither of their classes extend the other. Two types from the same class are disjoint, from rule (DTYPE-CTX), if two contexts in the same argument position are provably disjoint. Rule (DTYPE-SUB) states that subtyping preserves the disjointness relation.

A consequence of these rules, indeed the guiding principle behind them, is the following lemma.

LEMMA 7.1. *If $E; B \vdash t \,\#\, t'$, then there is no t'' for which both $E; B \vdash t'' \leq t$ and $E; B \vdash t'' \leq t'$ hold.*

Different rules would be required for a language with multiple inheritance and/or interfaces. The basic modification is to change the rule (DTYPE-CLASS) to apply only when the two classes or interfaces c and c' have no subclasses or subinterfaces in common. For a language with multiple inheritance in the case where an open world assumption is made, namely that we are only ever reasoning about a part of a program, no rule comparable to (DTYPE-CLASS) would exist.

Disjointness of Effect Shape. The disjointness of effect shape ultimately relies the tree-shaped nesting of objects, using the shapes depicted in Figure 5 to guide our design.

Disjointness of Effect Shape I

$$\text{(DFX-}\emptyset\text{)}\quad \frac{E; B \vdash \phi}{E; B \vdash \emptyset \,\#\, \phi} \qquad \text{(DFX-SUB)}\quad \frac{E; B \vdash \phi \,\#\, \phi' \quad E; B \vdash \phi'' \sqsubseteq \phi'}{E; B \vdash \phi \,\#\, \phi''}$$

$$\text{(DFX-UNION)}\quad \frac{E; B \vdash \phi \,\#\, \phi'' \quad E; B \vdash \phi' \,\#\, \phi''}{E; B \vdash \phi \cup \phi' \,\#\, \phi''}$$

The first three rules, (DFX-$\emptyset$), (DFX-SUB), and (DFX-UNION), are structural, derived from simple properties of sets. The others are more interesting.

Disjointness of Effect Shape II

$$\text{(DFX-NEQ)}\quad \frac{E; B \vdash p \quad n, m \geq 0 \quad n \neq m}{E; B \vdash p.n \,\#\, p.m} \qquad \text{(DFX-BAND0)}\quad \frac{E; B \vdash p \,\#\, q}{E; B \vdash p.0 \,\#\, q.0}$$

$$\text{(DFX-BANDUNDER1)}\quad \frac{E; B \vdash p \prec^{+} q \quad n \geq 0}{E; B \vdash q.n \,\#\, \texttt{under}(p.n)} \qquad \text{(DFX-BANDUNDER2)}\quad \frac{E; B \vdash p \prec^{*} q \quad n \geq 0}{E; B \vdash q.n \,\#\, \texttt{under}(p.n{+}1)}$$

$$\text{(DFX-INC)}\quad \frac{E; B \vdash p.n \,\#\, q.m}{E; B \vdash p.n{+}1 \,\#\, q.m{+}1} \qquad \text{(DFX-ABANDAPART)}\quad \frac{E; B \vdash p \prec q \quad m, n \geq 0 \quad m \neq n{+}1}{E; B \vdash p.n \,\#\, q.m}$$

$$\text{(DFX-UNDERAPART)}\quad \frac{E; B \vdash p \prec p' \quad E; B \vdash q \prec p' \quad \text{for some } p' \quad E; B \vdash p \,\#\, q}{E; B \vdash \texttt{under}(p) \,\#\, \texttt{under}(q)}$$

Rule (DFX-NEQ) states that different bands stemming from the same context are disjoint. If two contexts are not equal, then from rule (DFX-BAND0) the shape containing just one context is disjoint from that containing just the other. Rules (DFX-BANDUNDER1) and (DFX-BANDUNDER2) give conditions under which a band and an under shape are disjoint. If the bands at two levels are disjoint, from rule (DFX-INC) the two bands one step deeper are disjoint. Knowing that a context is directly inside another one allows more precision, as in rule (DFX-ABANDAPART). The final rule, (DFX-UNDERAPART), captures that the under effects of two disjoint objects which have the same owner are disjoint. This corresponds to the fact that contexts form a tree and that two subtrees rooted at disjoint nodes having the same parent are disjoint.

Properties. The following interesting fact can be derived from these rules. Ultimately this states that the internals of two distinct objects are completely disjoint.

LEMMA 7.2. *Assume $E; B \vdash \texttt{under}(p) \,\#\, \texttt{under}(q)$. Then $E; B \vdash p.n \,\#\, q.m$ for $n, m \geq 0$.*

The theorems which follow when combined with the Extension Lemma (Lemma 5.2) imply that a disjointness judgement holds regardless of the bindings of free variables. Thus these theorems hold for any program configuration which satisfies the initial assumptions in E, under any valid extension to E and/or B. The theorems all stem from the following, which states that two contexts which we can prove to be disjoint can never be equal:

THEOREM 7.3 (CONTEXT DISJOINTNESS). *Whenever $E; B \vdash p \,\#\, q$ holds, $E; B \vdash p = q$ is impossible.*

7.1 Disjoint types restrict aliasing

If we can prove that two types are disjoint, then it is impossible for two variables having these types to be aliases:

THEOREM 7.4 (TYPE DISJOINTNESS). *If $E; B \vdash t \,\#\, t'$, then for $x : t \in E$ and $y : t' \in E$, there is no $\iota \in dom(E)$ for which $x = \iota \in B$ and $y = \iota \in B$.*

A useful corollary of this theorem is that the evaluation of expressions with disjoint types never produces results which are aliases:

COROLLARY 7.5. *Whenever $E; B \vdash t \,\#\, t'$, $E; B \vdash e : t$, and $E; B \vdash e' : t'$, then for any H such that $E \vdash H$, if $\langle H; B; e\rangle \xrightarrow{\psi} \langle H'; v\rangle$ and $\langle H; B; e'\rangle \xrightarrow{\psi'} \langle H''; v'\rangle$, then either $v = \texttt{null}$ or $v \neq v'$.*

Corollary 7.5 is the formal justification of why we can conclude that `shared` and `encaps` from Section 2.2 will never be aliases.

Now consider the declarations:

```
List<world,world> shared;
List<this, world> encaps;
List<this, this>  local;
```

Since we can prove that $E \vdash \texttt{this} \,\#\, \texttt{world}$, we can conclude that `encaps`, `shared`, and `local` are not aliases — their types are provably disjoint. Furthermore we can also conclude that the references to internal representation, `shared.head` and `encaps.head`, cannot be aliases. Indeed none of

$$\left\{ \begin{array}{l} \texttt{shared.head}, \texttt{shared.head.next}, \\ \texttt{shared.head.next.next}, \ldots \end{array} \right\}$$

can be an alias for any of

$$\left\{ \begin{array}{l} \texttt{encaps.head}, \texttt{encaps.head.next}, \\ \texttt{encaps.head.next.next}, \ldots \end{array} \right\}.$$

Also, `shared.head.data` and `local.head.data` can never be aliases. On the other hand,

$$\left\{ \begin{array}{l} \texttt{shared.head.data, encaps.head.data,} \\ \texttt{shared.head.next.data, encaps.head.next.data,} \ldots \end{array} \right\},$$

all have the same type, namely $\texttt{Data}\langle\texttt{data}\rangle$, and therefore may be aliases.

Application to formal reasoning. When reasoning about object-oriented programs, *e.g.,* when calculating weakest preconditions, an explicit alias test is frequently inserted into assertion statements [24]. Such alias tests tend to cause a blow up in the complexity of the consequent verification. Simple class-based alias tests reduce the complexity somewhat, so we expect that our alias test will offer further advantage.

7.2 Disjoint effects restrict interference

The soundness of effects disjointness again amounts to showing that effects shapes do not overlap when our judgements assert that they are disjoint. We show this by considering the underlying basic shape of an effect shape.

Firstly, we can show that the basic shapes underlying the effects shapes are provably disjoint, and that, for basic shapes, the disjointness relation $E; B \vdash \phi \# \phi'$ can be interpreted as the disjointness of the underlying sets.

LEMMA 7.6. *If* $E;B \vdash \phi \# \phi'$, *then* $E;B \vdash \phi\downarrow_{E;B} \# \phi'\downarrow_{E;B}$. *Furthermore, if* ϕ *and* ϕ' *are basic, then* $\phi \cap \phi' = \emptyset$.

Next, as a consequence of these results and Lemma 5.2, we obtain the following theorem which states that the sets of contexts denoted by disjoint effects are disjoint in all possible configurations.

THEOREM 7.7 (DISJOINT SHAPES). *If* $E; B \vdash \phi \# \phi'$, $E' \gg E$, $B' \gg B$ *and* $E'; B' \vdash \diamond$, *then* $\phi\downarrow_{E';B'} \cap \phi'\downarrow_{E';B'} = \emptyset$.

We now define when two effects are non-interfering. This definition is based on simple notions of data dependence: avoiding swapping a write with a read or write to the same location [2].

Non-interference

(NON-INTERFERENCE)
$$\frac{E; B \vdash \phi_w \#(\phi'_r \cup \phi'_w) \quad E; B \vdash \phi'_w \#(\phi_r \cup \phi_w)}{E; B \vdash (\texttt{rd}\ \phi_r\ \texttt{wr}\ \phi_w) \#(\texttt{rd}\ \phi'_r\ \texttt{wr}\ \phi'_w)}$$

The following theorem states that non-interfering effects imply non-interfering execution, *i.e.,* that the order of evaluation of two expressions with disjoint effects is immaterial, producing the same results and the same heap.

THEOREM 7.8. *Assume* $E; B \vdash e : t!\psi$, $E; B \vdash e' : t'!\psi'$, $E; B \vdash \psi \# \psi'$, *and* $E \vdash H$. *Then, if*

$$\langle H; B; e\rangle \xrightarrow{\psi''} \langle H'; v\rangle \text{ and } \langle H'; B; e'\rangle \xrightarrow{\psi'''} \langle H''; v'\rangle,$$

there exists a H''' *such that*

$$\langle H; B; e'\rangle \xrightarrow{\psi'''} \langle H'''; v'\rangle \text{ and } \langle H'''; B; e\rangle \xrightarrow{\psi''} \langle H''; v\rangle.$$

The proof makes use of the fact that our semantics are based on relations which capture the nondeterministic choice of names for the locations of newly created objects.

Examples. We can check whether the evaluation of an expression e will affect an object z, by considering its effect ψ. If we can prove that $E \vdash \psi \# \texttt{rd}\ z$, then we know that e does not affect the fields of z. If we can prove the stronger judgement that $E \vdash \psi \# \texttt{rd}\ \texttt{under}(z)$, then we know that e affects neither z nor its representation.

We now apply the above ideas to the following:

```
List<p,world> list = new List<p,world>;
Iterator<list,world> iter = list.makeIterator();
```

Examining the effects of each of the methods, modulo parameter and `this` bindings, we obtain:

method	effect
`list.add`	`wr under(list)`
`iter.next`	`rd list.1 wr iter`
`iter.elem`	`rd list.1`

For an appropriate E, we know that $E \vdash \texttt{iter} \prec \texttt{list}$, using (IN-REP). Hence $E \vdash \texttt{iter} \# \texttt{list}$. Thus we can conclude that $E \vdash \texttt{iter} \sqsubseteq \texttt{list.1} \sqsubseteq \texttt{under(list)}$, using the subshape rules.

Our system reflects the fact that changes to `list` may affect `iter` (which of course is also visible by inspection of the method bodies) . For example, we cannot show that the method call `list.add` does not interfere with `iter`. This is because $E \vdash \texttt{wr under(list)} \# \texttt{rd under(iter)}$ *cannot* be proven due to the inclusion $E \vdash \texttt{under(iter)} \sqsubseteq \texttt{under(list)}$.

Our system also reflects the fact that calling the methods `next` or `elem` on `iter` do not affect `list`. That is, we can show $E \vdash (\texttt{rd list.1 wr iter}) \# \texttt{rd list}$.

Of course, we cannot show that `iter.next()` does not affect the list *and* its representation, since `iter` is a part of the representation of `list`. That is, it is impossible to show $E \vdash \texttt{rd list.1 wr iter} \# \texttt{rd under(list)}$.

We now move on to a different example. If we can show that two lists are independent, then we can show that operations on their respective internal representations will not interfere with each other. This can be used to perform potential optimisations, such as loop fusion.

Consider the following code:

```
List<p,world> list1;
List<q,world> list2;
....
for (i = 0; i < 10; i++) {
  list1.add(new Data<world>(i));    // exp1
}
for (i = 0; i < 10; i++) {
  list2.add(new Data<world>(i));    // exp2
}
```

Firstly, assuming that $E \vdash p \# q$, $p \prec p'$ and $q \prec p'$ for some p', we can prove that $E \vdash \texttt{List}\langle p, \texttt{world}\rangle \# \texttt{List}\langle q, \texttt{world}\rangle$, and hence deduce that `list1` and `list2` cannot be aliases. Next, we can infer that the effect produced by the expression marked `exp1` is `wr under(list1)` and that the effect of expression `exp2` is `wr under(list2)`. Finally, obtaining $E \vdash \texttt{wr list1} \# \texttt{wr list2}$, we demonstrate that the two computations `exp1` and `exp2` do not interfere. This then permits the following loop fusion:

```
for (i = 0; i < 10; i++) {
  list1.add(new Data<world>(i));
  list2.add(new Data<world>(i));
}
```

These examples demonstrate that our type system may help improve tools ranging from program understanding to optimisation.

8. RELATED WORK

The Geneva Convention on the Treatment of Object Aliasing [33] stressed the need for better treatment of aliasing in object-oriented programming. Early approaches following this lead include Islands [32] and Balloons [4]. These focused on *fully encapsulated* objects, where all objects that an object could access were not accessible outside the object. Moving objects across encapsulation boundaries either required copy-assignment (which defeats object identity) or destructive-reads. Whereas the latter can successfully be replaced by ordinary reads, with the help of Boyland's analysis [13], an odd programming style is still required to use such "slippery variables." Other approaches use uniqueness [41, 32, 13, 47, 6], but these also require destructive reads.

Some systems approach the problems caused by aliasing using *read-only* references [40, 32, 34, 47, 35, 44, 54, 15, 51], generally by employing a fixed version of C++'s `const` [52] which limits computation to only reading the transitive state of a reference. Most of these papers describe neither how read-only references help reason about programs nor how they help write more robust software.

Boyland, Noble and Retert present a capabilities system in which to uniformly deal with read, write, and ownership capabilites, also capturing notions such as uniqueness and borrowing in the process [15]. A capability is attached to a reference to limit what can be done with the reference. *Exclusive capabilities* refuse all other references from employing the capability, giving the system a neat duality. The system can express many concepts with just 4 capabilites — read, write, identity, and ownership — of which an exclusive capability exists for all but the last. Unfortunately, the capabilites apply only to references and not what they refers to, nor is it clear how to use the semantics, especially since the exclusive effects exhibit non-local behaviour.

The most sophisticated approach for dealing with aliasing in object-oriented programming is Noble, Vitek, and Potter's *Flexible alias protection* [47]. This proposal introduced a number of important new ideas. Most prominent was the enforcible notion of representation, which we employ here. Secondly, classes were parameterised by what we now call contexts, allowing classes to be used in different contexts. (Classes were also generic.) In addition, there was a mode call `arg` which was used to enforce a property called *argument independence*, *i.e.,* `arg` limited the use of a reference to only methods which did not read mutable state. The system, however, lacked inheritance, nor was it formalised, and a gap remained between the system and the properties it could give to a formal reasoner.

Ownership types were devised to formalise the core of Flexible alias protection [22, 21]. Apart from providing a formal system, this work contributed the notion of owner, required both to give a type to self and for some idioms such as allowing an arbitrary object graph as a part of an object's representation, and a formal demonstration of the *owners-as-dominators* property. Ownership types are explored further in the first author's thesis [20], mostly in the context of Abadi and Cardelli's object calculus [1].

Other systems similar to ownership types have been developed, sometimes independently. Universes is an ownership types system which forgoes parameterised classes and employs instead pervasive read-only references [44]. This a part of a modular reasoning system for Java [43] and has been used to specify useful frame properties in JML [45]. Confined types adopt package-level ownership to provide better security in Java [9]. Boypati and Rinard [12] employ object ownership to enable race-free Java programs, and more recently (with Lee) for preventing deadlocks [11]. Their type systems offer only shallow ownership and a different form of effect, which is sufficient for the problem they address, but not enough to obtain the results presented here. Boypati, Lee and Rinard's work extends prior systems such as Guava [5], which also has a notion of thread ownership, and Flanagan and Freund's [29] which parameterises classes by locks [29]. This last work incidentally adapts ideas from outside the object-oriented world [28]. Recently, Aldrich, Kostadinov and Chambers developed AliasJava which includes a shallow notion of object ownership, uniqueness and borrowing for method arguments [3]. They also include a sophistocated algorithm for inferring alias annotations which should surely be adaptable to the system presented here. In addition they have some results which indicate that their system is practical. Banerjee and Naumann have demonstrated the advantage of having confinement properties in Java programs [7]. While their language does not include a specific means for denoting representation, as ownership types provide, they are able to reason about incidental representation. Using a different semantic approach, they are able to achieve somewhat deeper results, including *representation independence*. This encouraging result is a formal demonstration that confined representations can be replaced with semantically equivalent ones, without affecting the surrounding program.

The seminal work dealing with interference was Reynolds' *syntactic control of interference* proposal which addressed the issue of parameter aliasing in Algol programs [50]. This proposal was, however, overly conservative. Effects systems overcame these limitations, initially by requiring explicit annotations [39], then through program analysis [53]. The later approach developed into region-based memory management for the ML programming language [55]. There have been many refinements, improvements and variations to region-based memory management, but most were for functional programming languages and none considered object-oriented programs.

Back in the object-oriented world, Leino described how *modifies* clauses should interact with subclassing to enable modular specifications [38]. His approach crystalised the key to modular effects specifications, although it applies only to a finite collection of regions (called data groups). Greenhouse and Boyland followed this lead when designing their object-oriented effects system, which specifies both read and write effects, employs uniqueness and enables a notion of representation [30]. There are a number of differences between the system described in this paper and Greenhouse and Boyland's. The first main difference is that they employ uniqueness instead of ownership for alias control, which results in a different style of programming. (Elsewhere Boyland argues that an object-oriented effects system must employ either uniqueness or ownership [14].) Secondly, our effect shapes slice across different depths of an object graph (as illustrated in Figure 5), whereas theirs slice each object into groups of fields. This kind of effects shape can be added to our system, but ours cannot be readily added to theirs because they lack ownership. The final main difference is that our system is parameterised by context/region information, whereas theirs is not. Merging the best features of both will be fruitful. Yates presents a type-and-effect system for Java [59]. He employs a global collection of regions but also allows effect masking for expressions, *i.e.,* the trimming of effects which operate on objects that cannot accessed outside of the scope of some expression. Classes have only what is in effect an owner parameter, which cannot, however, be exploited internally as our owner can, but there are no other parameters nor any notion of representation.

Our type disjointness test in part resembles that of Diwan, McKinley and Moss [27], except they do not have the additional ownership information and the nesting of objects, as this is not available in existing programming languages. More sophisticated aliasing analyses exist, and could be combined with ours for more precision.

The calculus of capabilities [23] uses linearity and uniqueness (of region) to enable non-stack-based, region-based memory management. The techniques they use for uniqueness (of region passed to a method) could be added here to enhance our disjointness tests, should we wish to make assumptions about disjointness in class headers or method parameter lists. Leavens and Antropova do so using dynamic dispatch to eliminate parameter aliasing [36]. Their aim is the simplify method specifications, since definite guarantees of parameter aliasing become available. Their system is compatible with ours and would indeed enhance our disjointness test considerably, since they provide a very fine-grained form of disjointness which we do not provide: they can express that two variables of the same type are definitely or definitely not aliases.

The programming language Cyclone [31], a C variant with region-based memory management, has a notion of ownership, though not object ownership, and an *outlives* ordering, similar to our nesting, which enables memory to be deallocated in a provably safe manner avoiding dangling pointers. Subtyping in Cyclone permits change of ownership, which is unsound in our system as we demonstrated in Section 5.1. This ultimately is because our types are based on classes, whereas in Cyclone types are based on more low-level stuctures. The type theory underlying Cyclone could serve as a fundamental semantics for our language, just as the work in the first author's thesis does. The designers of Cyclone expended much effort reducing syntactic burden the annotations impose on a programmer. Similar techniques could apply here; indeed some indications in this direction were made in the first author's thesis (for classes which are also generic) [20]. Vault [25] extend C in a manner similar to Cyclone, though relying more on linearity instead of regions. Following this style requires quite low-level annotations, though some of the techniques may spill over into an object-oriented programming. Neither Cyclone nor Vault deal with objects and the consequent subtyping.

9. CONCLUSIONS AND FUTURE WORK

We have extended previous work on ownership types to support inheritance and dynamic aliases, while maintaining the *owners-as-dominators* property and the consequent strong form of encapsulation. To the resulting ownership type system, we have added computational effects in a manner exploiting the strong encapsulation. Finally, we have applied ownership types to reasoning about the absence of aliasing, and the effects system to reason about the non-interference of computation. Our prototype implementation is being extending to encorporate the features described here for a more complete programming language.

There are two main directions for future work. Firstly, we would like to see a closer integration with a specification language, such as JML [37], and an associated formal reasoning system [57]. This we expect would reduce the burden of reasoning about applications such as Java Card [18]. Indeed, this has in part been done using the Universes ownership types system [45]. Secondly, we wish to extend our programming language further. Firstly, we would like to include effects masking to localise effects to within expressions. This will also opens the way for stack-allocable objects, or at least objects which can be allocated/deallocated in a stack-based manner, *e.g.,* like scoped memories used in Real-time Java [10]. In addition, by allowing context parameterisation on methods, we will enable a notion of borrowing [13, 41, 32, 5, 15], which is useful for some applications.

Acknowledgments. Thanks to James Noble and Tobias Wrigstad for their useful comments on earlier versions of this paper, and to the anonymous referees who suggested numerous improvements including the StockQuoteHandler example.

10. REFERENCES

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

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

[3] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias annotations for program understanding. In *OOPSLA*, November 2002.

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

[5] David F. Bacon, Robert E. Strom, and Ashis Tarafdar. Guava: a dialect of Java without data races. In *OOPSLA Proceedings*, pages 382–400, 2000.

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

[7] Anindya Banerjee and David A. Naumann. Representation independence, confinement, and access control. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages (POPL'02)*, Portland, Oregon, January 2002.

[8] Bruno Blanchet. Escape analysis for object-oriented languages: Application to Java. In *OOPSLA Proceedings*, October 1999.

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

[10] Greg Bollella, James Gosling, Benjamin Brosgol, Peter Dibble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java.* Addison-Wesley, 2000.

[11] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. A type system for preventing data races and

deadlocks in Java programs. In *OOPSLA*, November 2002.

[12] Chandrasekhar Boyapati and Martin Rinard. A parameterized type system for race-free Java programs. In *OOPSLA Proceedings*, 2001.

[13] John Boyland. Alias burying: Unique variables without destructive reads. *Software — Practice and Experience*, 31(6):533–553, May 2001.

[14] John Boyland. The interdependence of effects and uniqueness. In *3rd Workshop on Formal Techniques for Java Programs*, June 2001.

[15] John Boyland, James Noble, and William Retert. Capabilities for Sharing: A Generalization of Uniqueness and Read-Only. In *ECOOP Proceedings*, June 2001.

[16] Ciarán Bryce and Chrislain Razafimahefa. An approach to safe object sharing. In *OOPSLA Proceedings*, October 1999.

[17] 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.

[18] Z. Chen. *Java Card Technology for Smart Cards*. The Java Series. Addison-Wesley, 2000.

[19] David Clarke. An object calculus with ownership and containment. In *Foundations of Object-oriented Programming (FOOL8)*, London, January 2001. Available from `http://www.cs.williams.edu/∼kim/FOOL/FOOL8.html`.

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

[21] David Clarke, James Noble, and John Potter. Simple ownership types for object containment. In *ECOOP Proceedings*, June 2001.

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

[23] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *1999 Symposium on Principles of Programming Languages*, 1999.

[24] Frank S. de Boer. A WP-calculus for OO. In *Foundations of Software Science and Computation Structures (FOSSACS'99)*, volume 1578 of *LNCS*, 1999.

[25] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 59–69, June 2001.

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

[27] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. Type-based alias analysis. In *Proceedigns of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation*, June 1998.

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

[29] Cormac Flanagan and Stephen Freund. Type-based race detection for Java. In *ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, June 2000.

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

[31] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in cyclone. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, June 2002.

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

[33] 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.

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

[35] Günter Kniesel and Dirk Theisen. JAC – Java with transitive readonly access control. In *Intercontinental Workshop on Aliasing in Object-Oriented Systems*, At ECOOP'99, Lisbon, Portugal, June 1999.

[36] Gary T. Leavens and Olga Antropova. ACL — eliminating parameter aliasing with dynamic dispatch. Technical Report 98-08a, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, February 1999.

[37] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, Boston, 1999.

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

[39] 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.

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

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

[42] Greg Morrisett and Robert Harper. Semantics of memory management for polymorphic languages. In Andrew D. Gordon and Andrew M. Pitts, editors, *Higher Order Operational Techniques in Semantics*, volume 12 of *Publications of the Newton Institute*, pages 175–226. Cambridge University Press, 1998.

[43] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*. PhD thesis, FernUniversität Hagen, 2001.

[44] 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.

[45] Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular specification of frame properties in JML. Technical Report TR-02-02, Department of Computer Science, Iowa State University, February 2002.

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

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

[48] Martin Odersky and Christoph Zenger. Nested types. In *Foundations of Object-oriented Programming (FOOL8)*, London, January 2001. Available from `http://www.cs.williams.edu/~kim/FOOL/FOOL8.html`.

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

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

[51] Mats Skoglund and Tobias Wrigstad. Alias control with read-only references. In *Sixth Conference on Computer Science and Informatics*, March 2002.

[52] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.

[53] J.-P Talpin and P. Jouvelot. Polymorphic type, region, and effect inference. *Journal of Functional Programming*, 2(3):245–271, July 1992.

[54] Dirk Theisen. Enhancing encapsulation in OOP: A practical approach. Master's thesis, Institut für Informatik III, Römerst. 164, D-53117 Bonn, Universität Bonn.

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

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

[57] Joachim van den Berg and Bart Jacobs. The LOOP compiler for Java and JML. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Software (TACAS)*, LNCS, pages 299–312, 2031, 2001. Springer-Verlag.

[58] Alan Wills. Reasoning about aliasing. In *ECOOP Proceedings*, 1993.

[59] Bennett Norton Yates. A type-and-effect system for encapsulating memory in Java. Master's thesis, Department of Computer and Information Science and the Graduate School of the University of Oregon, August 1999.