Comparing Universes and Existential Ownership Types

Nicholas Cameron
Victoria University of Wellington
ncameron@ecs.vuw.ac.nz

Werner Dietl
ETH Zurich
Werner.Dietl@inf.ethz.ch

ABSTRACT

Ownership types and Universe types are two type systems used to structure the heap and enforce encapsulation disciplines. The parametricity of ownership types allows a finer-grained description of heap topologies, whereas the flexibility of `any` references in Universe types allows sharing between data structures. No direct encoding of one type system in the other has been possible.

Parametric ownership has recently been extended with existential quantification of contexts. We formalise such a language and give a formal translation between programs written in this language and using Universe types. We show that this translation is sound and complete.

1. INTRODUCTION

Parametric ownership types [13] and Universes [25, 17] are two ownership type systems which describe an ownership hierarchy and statically check that this hierarchy is maintained; that is, they include a descriptive part. *How* the two systems describe this hierarchy is different. Both systems also provide (different) encapsulation properties based on this hierarchy; a prescriptive part.

Ownership types can describe fine-grained heap topologies, whereas Universe types are more flexible and easier to use. No direct encoding of one type system in the other has been possible: the abstraction provided by `any` references in Universes could not be modeled with parametric ownership types.

Recently, parametric ownership has been extended with existential quantification of contexts [8, 6]. This extension, called Jo∃, provides the possibility to abstract from concrete ownership information — similarly to `any` references in Universe types.

In this paper, we show that the descriptive parts of the Universe type system [14] and a variant of Jo∃, which we call Jo$\exists^-$, are equivalent (though note that full Jo∃ is more expressive than Universes). We formalise this correspondence as encodings between the two systems. We have proved that the encodings from Universes to Jo$\exists^-$ and from Jo$\exists^-$ to Universes are sound; thus, we have shown that the two systems are equivalent with respect to type checking. As an intermediate step in the encoding we give an alternative formalisation of Universes which is closer to the underlying existential types model.

The outline of the rest of the paper is as follows. Sect. 2 gives a short summary of previous work on ownership type systems. Sect. 3 presents our formalisation of the Universe type system, and in Sect. 4 we present Jo$\exists^-$ and explain how it differs from Jo∃. In Sect. 5, we show a correspondence between these two ownership systems. Finally, in Sect. 6 we discuss our contributions and future work.

2. BACKGROUND — OWNERSHIP

Object ownership structures the heap hierarchically and allows for the control of aliasing and access between objects. Ownership has been successfully used in a variety of different contexts; for example, for program verification [22, 25, 26, 24], architecture description [1], thread synchronization [4, 20, 15], memory management [2, 5], and representation independence [3].

There are many flavours of ownership. These systems have in common the concept of an ownership topology, but differ in how it is specified and enforced. Some common concepts are shared by all (or most) systems: each object is owned by at most one other object, called its *owner*; the set of all objects with the same owner is called a *context*; objects without an owner are located in a *root context*, the root of the ownership tree.

Conceptually, ownership systems can be split into two parts: an ownership topology and an encapsulation discipline. The ownership topology describes the hierarchical structure of the heap, whereas an encapsulation discipline enforces aliasing and access restrictions. Most ownership systems enforce a topology and an encapsulation discipline at once. In this paper, we will only be concerned with the descriptive aspect of ownership systems, that is, how the ownership topology is enforced by the type system.

Parametric ownership types [27, 13, 11, 12] parameterise classes and references with owner parameters. They enforce the *owner-as-dominator* encapsulation discipline: all reference chains from the root context to an object in a different context must go through that object's owner. This restriction of aliasing is of benefit in some applications of ownership, for example, memory management, garbage collection, and representation independence.

The Universe type system [25, 16] is an alternative own-

IWACO '09, July 6 2009, Genova, Italy
Copyright 2009 ACM 978-1-60558-546-8/09/07 ...$10.00.

$$\begin{array}{lcll}
\mathtt{e} & ::= & \mathtt{null} \mid \mathtt{x} \mid \mathtt{e.f} \mid \mathtt{e.f = e} \mid & \textit{expressions} \\
 & & \mathtt{e.m}(\overline{\mathtt{e}}) \mid \mathtt{new\ S} & \\
 & & & \\
\mathtt{Q} & ::= & \mathtt{class\ C} \lhd \mathtt{D}\ \{\overline{\mathtt{S\,f}};\ \overline{\mathtt{W}}\} & \textit{class declarations} \\
\mathtt{W} & ::= & \mathtt{S\,m}(\overline{\mathtt{S\,x}})\ \mathtt{\{return\ e;\}} & \textit{method declarations} \\
 & & & \\
\mathtt{u}_s & ::= & \mathtt{rep} \mid \mathtt{peer} \mid \mathtt{any} & \textit{source Universe modifiers} \\
\mathtt{u} & ::= & \mathtt{u}_s \mid \mathtt{lost} \mid \mathtt{self} & \textit{Universe modifiers} \\
\mathtt{S} & ::= & \mathtt{u}_s\ \mathtt{C} & \textit{source types} \\
\mathtt{T}, \mathtt{U} & ::= & \mathtt{u\ C} & \textit{types} \\
 & & & \\
\Gamma & ::= & \overline{\mathtt{x{:}T}} & \textit{variable environments} \\
 & & & \\
\mathtt{x}, \mathtt{y}, \mathtt{this} & & & \textit{variables} \\
\mathtt{C}, \mathtt{D} & & & \textit{classes} \\
\mathtt{f} & & & \textit{field names} \\
\mathtt{m} & & & \textit{method names}
\end{array}$$

Figure 1: Universes: syntax.

$$\frac{}{\vdash_u \mathtt{self} \leq \mathtt{peer}}\ \text{(UO-Self)} \qquad \frac{}{\vdash_u \mathtt{peer} \leq \mathtt{lost}}\ \text{(UO-Peer)}$$

$$\frac{}{\vdash_u \mathtt{rep} \leq \mathtt{lost}}\ \text{(UO-Rep)} \qquad \frac{}{\vdash_u \mathtt{lost} \leq \mathtt{any}}\ \text{(UO-Any)}$$

$$\frac{}{\vdash_u \mathtt{u} \leq \mathtt{u}}\ \text{(UO-Reflex)} \qquad \frac{\vdash_u \mathtt{u}_1 \leq \mathtt{u}_3 \qquad \vdash_u \mathtt{u}_3 \leq \mathtt{u}_2}{\vdash_u \mathtt{u}_1 \leq \mathtt{u}_2}\ \text{(UO-Trans)}$$

$$\frac{\mathtt{class\ C} \lhd \mathtt{D}\ \ldots}{\vdash_u \mathtt{u\ C} <: \mathtt{u\ D}}\ \text{(US-Sub-Class)} \qquad \frac{\vdash_u \mathtt{u} \leq \mathtt{u}'}{\vdash_u \mathtt{u\ C} <: \mathtt{u}'\ \mathtt{C}}\ \text{(US-Env)}$$

$$\frac{}{\vdash_u \mathtt{T} <: \mathtt{T}}\ \text{(US-Reflex)} \qquad \frac{\vdash_u \mathtt{T}_1 <: \mathtt{T}_3 \qquad \vdash_u \mathtt{T}_3 <: \mathtt{T}_2}{\vdash_u \mathtt{T}_1 <: \mathtt{T}_2}\ \text{(US-Trans)}$$

Figure 2: Universes: ordering and subtyping.

		u₂				
	$\mathtt{u}_1 \rhd \mathtt{u}_2$	self	peer	rep	any	lost
	self	self	peer	rep	any	lost
	peer	lost	peer	lost	any	lost
$\mathtt{u}_1$	rep	lost	rep	lost	any	lost
	any	lost	lost	lost	any	lost
	lost	lost	lost	lost	any	lost

Figure 3: Universes: viewpoint adaptation.

ership type system aimed at modular formal verification of object-oriented software [22, 26, 17]. It enforces the *owner-as-modifier* encapsulation discipline: an object `o` may be referenced by any other object, but reference chains that do not pass through `o`'s owner must not be used to modify `o`. This discipline allows objects to control state changes and thereby enforce invariants of owned objects. A recent formalisation of the Universe type system [14] separates the ownership topology from the owner-as-modifier encapsulation discipline. We build on the topological system of this formalisation.

Previous work has studied an encoding of different ownership type systems in dependent classes [18]. However, no formal relationship between the different systems has been shown.

3. UNIVERSE TYPE SYSTEM

In this section, we present a formalisation of the Universe type system based on a previous formalisation [14]. It is a Java-like object-oriented programming language with much of the notation similar to Featherweight Java [19].

Fig. 1 presents the syntax of the programming language. We support the usual expressions: the `null` literal, reading variables `x` (which includes `this`), reading and updating the value of a field, invoking a method, and creating a new object. Class and method declarations are standard. We do not specify method purity in method declarations because we are only concerned with the topological system.

We distinguish between source Universe modifiers $\mathtt{u}_s$ and (internal) Universe modifiers `u` and also between source types `S` and (internal) types `T`. Source types appear in programs and are limited to `peer`, `rep`, and `any` Universe modifiers. These represent a reference to an object in the same context, to an owned object, and to an object with an arbitrary owner, respectively. The Universe modifiers `self` and `lost` may appear in typing derivations. The modifier `self` is used to denote a reference to the current object `this`; the modifier `lost` is used to express that no concrete ownership information (within the limits of the Universe modifier syntax) can be given.

Fig. 2 presents the subtyping rules for Universes. We first define an ordering relation on Universe modifiers. The `self` modifier is more concrete than `peer` (by UO-Self), both `peer` and `rep` are below `lost` (UO-Peer and UO-Rep), `lost` is below `any` (UO-Any), and the ordering is reflexive (UO-Reflex) and transitive (UO-Trans).

Subtyping follows the subclassing relationship introduced by the extends relation in the class declaration (US-Sub-Class) and Universe modifier ordering (US-Env).

Fig. 3 presents viewpoint adaptation. $\mathtt{u}_1 \rhd \mathtt{u}_2$ adapts the Universe modifier $\mathtt{u}_2$ from the point of view of $\mathtt{u}_1$ to the current viewpoint `this`. For example, accessing a field declared with a `peer` type through an expression that has a `rep` type results in a $\mathtt{rep} \rhd \mathtt{peer} = \mathtt{rep}$ type. We also adapt a type from the point of view of a Universe modifier, written as $\mathtt{u}_1 \rhd \mathtt{u}_2\ \mathtt{C}$, which adapts the Universe modifier $\mathtt{u}_2$ from the point of view of $\mathtt{u}_1$ to the current viewpoint `this` and leaves the class `C` unchanged, i.e., $\mathtt{u}_1 \rhd (\mathtt{u}_2\ \mathtt{C})$ is defined to be $(\mathtt{u}_1 \rhd \mathtt{u}_2)\ \mathtt{C}$.

Lookup functions for methods and fields and the definitions of well-formed types, environments, classes, and methods are straightforward and have been relegated to the accompanying technical report[7].

The expression typing rules are given in Fig. 4. The `null` literal can receive an arbitrary well-formed source type (by UT-Null)[1]. Object creation requires a well-formed type

[1] This is a slight modification from the formalisation in [14]

$$\frac{}{\Gamma \vdash_u \mathtt{x} : \Gamma(\mathtt{x})}\ \text{(UT-Var)} \qquad \frac{\vdash_u \mathtt{S}\ \text{OK}}{\Gamma \vdash_u \mathtt{null} : \mathtt{S}}\ \text{(UT-Null)}$$

$$\frac{\vdash_u \mathtt{u\ C}\ \text{OK} \quad \mathtt{u} \in \{\mathtt{rep,\ peer}\}}{\Gamma \vdash_u \mathtt{new\ u\ C} : \mathtt{u\ C}}\ \text{(UT-New)} \qquad \frac{\Gamma \vdash_u \mathtt{e} : \mathtt{u\ C} \quad \mathit{fType}(\mathtt{f}, \mathtt{C}) = \mathtt{T}}{\Gamma \vdash_u \mathtt{e.f} : \mathtt{u} \rhd \mathtt{T}}\ \text{(UT-Field)}$$

$$\frac{\Gamma \vdash_u \mathtt{e} : \mathtt{u\ C} \quad \mathit{fType}(\mathtt{f}, \mathtt{C}) = \mathtt{T} \quad \mathtt{u} \rhd \mathtt{T} \neq \mathtt{lost\ D} \qquad \Gamma \vdash_u \mathtt{e}' : \mathtt{U} \quad \vdash_u \mathtt{U} <: \mathtt{u} \rhd \mathtt{T}}{\Gamma \vdash_u \mathtt{e.f\ =\ e}' : \mathtt{u} \rhd \mathtt{T}}\ \text{(UT-Assign)}$$

$$\frac{\Gamma \vdash_u \mathtt{e} : \mathtt{u\ C} \quad \mathit{mType}(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{T}} \rightarrow \mathtt{T} \qquad \Gamma \vdash_u \overline{\mathtt{e} : \mathtt{U}} \quad \vdash_u \overline{\mathtt{U} <: \mathtt{u} \rhd \mathtt{T}} \quad \mathtt{lost} \notin \mathtt{u} \rhd \overline{\mathtt{T}}}{\Gamma \vdash_u \mathtt{e.m}(\overline{\mathtt{e}}) : \mathtt{u} \rhd \mathtt{T}}\ \text{(UT-Invk)}$$

Figure 4: Universes: expression typing rules.

that uses the `peer` or `rep` Universe modifier (UT-New); this ensures that objects can only be created in a statically known context. As described earlier, the declared type of a field `T` needs to be adapted to account for the type of the receiver expression `u C` (UT-Field). Similarly, the type of a field assignment (UT-Assign) is determined by viewpoint adapting the declared field type. A field assignment is forbidden if the viewpoint adaptation results in `lost` ownership information. Finally, a method invocation (UT-Invk) adapts the parameter and return types and ensures that no ownership information in the parameters was `lost`.

3.1 An Alternate Formalisation of Universes

To prove the equivalence of Universes and Jo$\exists^-$ we used an intermediate language whose syntax is that of Universes, but whose static semantics reflect Jo$\exists^-$ more closely than existing formalisations. In this alternate formalisation of Universes, our aim is for subtyping to contain as much information about a type's behaviour as possible; in particular, the premises which check for `lost` in the Universe type rules should be avoided (since these premises effectively constrain the set of supertypes which can be found for a type). To avoid these premises, subtyping must be made more restrictive; happily, this change makes subtyping closer to that of Jo$\exists^-$.

We believe that some aspects of this formalisation are cleaner than the existing formalisation: similar premises are not duplicated and the relation between types is encapsulated within the subtype relation, rather than involving the type rules. We show that the two versions of Universes are equivalent in Sect. 5.

Fig. 5 presents subclassing and alternate Universe subtyping and Universe modifier ordering. Subclassing follows directly from the class declarations written by the programmer.

$$\frac{}{\vdash_u \mathtt{C} \lhd \mathtt{C}}\ \text{(USC-Reflex)} \qquad \frac{\vdash_u \mathtt{C}_1 \lhd \mathtt{C}_3 \quad \vdash_u \mathtt{C}_3 \lhd \mathtt{C}_2}{\vdash_u \mathtt{C}_1 \lhd \mathtt{C}_2}\ \text{(USC-Trans)}$$

$$\frac{\mathtt{class\ C} \lhd \mathtt{D\ \ldots}}{\vdash_u \mathtt{C} \lhd \mathtt{D}}\ \text{(USC-Sub-Class)} \qquad \frac{\vdash_u \mathtt{C} \lhd \mathtt{D} \quad \vdash_{u'} \mathtt{u} \leq \mathtt{u}'}{\vdash_{u'} \mathtt{u\ C} <: \mathtt{u}'\ \mathtt{D}}\ \text{(UAS-Env)}$$

$$\frac{\mathtt{u} \neq \mathtt{lost}}{\vdash_{u'} \mathtt{u} \leq \mathtt{u}}\ \text{(UAO-Reflex)} \qquad \frac{}{\vdash_{u'} \mathtt{self} \leq \mathtt{peer}}\ \text{(UAO-Self-Peer)} \qquad \frac{}{\vdash_{u'} \mathtt{u} \leq \mathtt{any}}\ \text{(UAO-Any)}$$

Figure 5: Universes: subclassing and alternate subtyping and ordering.

Alternate Universe modifier ordering forbids `lost` as the larger element. The ordering is only reflexive if the modifier is not `lost` (UAO-Reflex). Otherwise it is the same as the original Universe ordering: the `self` modifier is below `peer` (UAO-Self-Peer) and all modifiers are below `any` (UAO-Any).

Subtyping is given by UAS-Env and combines subclassing with Universe ordering.

Fig. 6 presents our alternate expression typing rules. Rules UAT-Var, UAT-Null, and UAT-New are unchanged from their UT-... equivalents. UAT-Assign and UAT-Invk are missing the check on `lost` since this is handled in the subtyping. All rules which look up and adapt a type use the *close* function to ensure that `lost` does not appear in any type assigned to an expression. This does not affect type checking because, under the alternate subtyping rules, an expression with `any` type can be used in all the places a `lost` type could and no more. The motivation for this change is simply that it matches Jo$\exists^-$ more closely.

However, note that we could not simply remove `lost` from the formalisation, i.e., by substituting `lost` by `any` in the viewpoint adaptation function (Fig. 3), without strengthening the type rules accordingly. Formalisations of the Universe type system without a `lost` modifier [17, 16] need additional checks for field updates and method calls to ensure type soundness. For example, consider a field with declared type `peer Object` that is accessed through an `any` reference. The viewpoint adapted type is `lost Object` and therefore an update is forbidden. If viewpoint adaptation were to return `any Object`, we could use an arbitrary reference as right-hand side of the field update and break type soundness. We follow [14] in using a separate `lost` modifier, we can therefore identify safe updates and method calls by viewpoint adaptation and thus use the simple type rules in Fig. 6.

4. PARAMETRIC OWNERSHIP AND Jo$\exists$

In parametric ownership systems [13], contexts are passed around a program as parameters to types. These context parameters describe parts of the heap topology in relative terms. By allowing contexts other than an object's owner to be named within a class, disparate parts of the heap can

where `null` could take any type (`T` as opposed to `S`). This is a minor change and does not affect the formalism that much. The only effect is that in assignments, `null` cannot be assigned to fields where no object could be assigned. We believe this is sensible; it reflects other formalisations of Universe types [16].

$$\frac{}{\Gamma \vdash_{u'} \mathtt{x} : \Gamma(\mathtt{x})}\ \text{(UAT-Var)} \qquad \frac{\vdash_u \mathtt{S}\ \text{OK}}{\Gamma \vdash_{u'} \mathtt{null} : \mathtt{S}}\ \text{(UAT-Null)}$$

$$\frac{\vdash_u \mathtt{u\ C}\ \text{OK} \qquad \mathtt{u} \in \{\mathtt{rep,\ peer}\}}{\Gamma \vdash_{u'} \mathtt{new\ u\ C} : \mathtt{u\ C}}\ \text{(UAT-New)} \qquad \frac{\Gamma \vdash_{u'} \mathtt{e} : \mathtt{u\ C} \qquad \mathit{fType}(\mathtt{f},\mathtt{C}) = \mathtt{T}}{\Gamma \vdash_{u'} \mathtt{e.f} : \mathit{close}(\mathtt{u} \rhd \mathtt{T})}\ \text{(UAT-Field)}$$

$$\frac{\Gamma \vdash_{u'} \mathtt{e} : \mathtt{u\ C} \qquad \mathit{fType}(\mathtt{f},\mathtt{C}) = \mathtt{T} \qquad \Gamma \vdash_{u'} \mathtt{e}' : \mathtt{U} \qquad \vdash_{u'} \mathtt{U} <: \mathtt{u} \rhd \mathtt{T}}{\Gamma \vdash_{u'} \mathtt{e.f\ =\ e}' : \mathit{close}(\mathtt{u} \rhd \mathtt{T})}\ \text{(UAT-Assign)}$$

$$\frac{\Gamma \vdash_{u'} \mathtt{e} : \mathtt{u\ C} \qquad \mathit{mType}(\mathtt{m},\mathtt{C}) = \overline{\mathtt{T}} \rightarrow \mathtt{T} \qquad \Gamma \vdash_{u'} \overline{\mathtt{e} : \mathtt{U}} \qquad \vdash_{u'} \overline{\mathtt{U} <: \mathtt{u} \rhd \mathtt{T}}}{\Gamma \vdash_{u'} \mathtt{e.m}(\overline{\mathtt{e}}) : \mathit{close}(\mathtt{u} \rhd \mathtt{T})}\ \text{(UAT-Invk)}$$

$$\mathit{close}(\mathtt{u\ C}) = \begin{cases} \mathtt{any\ C}, & \mathit{if}\ \mathtt{u} = \mathtt{lost} \\ \mathtt{u\ C}, & \mathit{otherwise} \end{cases}$$

Figure 6: Universes: alternative expression typing rules.

be used together in one class.

Classes are parameterised by formal context parameters and types by actual context parameters. The entities which can be used as an actual context varies from system to system, but include at least the current context, `this`, formal context parameters, and the root context, `world`. For example, a list can be declared as:

```
class List<owner, dOwner> {
  Object<dOwner> datum;
  List<owner, dOwner> next;
  Object<this> pf;
}
```

The formal context `owner` represents the owner of instantiations of the list class. The context `dOwner` is passed to the definition (without affecting the ownership topology) and is used as the owner of `datum`. The field `pf` is in the list's representation because it is owned by the list.

Ownership types are invariant with respect to their context parameters. That is, `Book<this>` is not a subtype of `Book<world>`, even though `this` is inside `world`. This invariance preserves owners across subtyping and is used to show soundness and enforce encapsulation properties.

4.1 Existential Quantification and Jo∃

In Java, existential types in the form of wildcards [28, 9] are used to implement subtype variance. Similarly, existential quantification of contexts can be used to give subtype variance in an ownership language [8].

Existential quantification has also been used to abstract contexts in ownership languages [11, 21]; and to support downcasting without storing runtime ownership information [29]. Wherever some form of variance is present in an ownership language [23, 10, 24, 14], the mechanisms for implementing it resemble implicit existential types [6].

e	::=	null | x | e.f | e.f = e | e.m($\overline{\mathtt{e}}$) | new C<a>	*expressions*
Q	::=	class C<owner> ◁ N {$\overline{\mathtt{T\,f;}}$ $\overline{\mathtt{W}}$}	*class declarations*
W	::=	T m($\overline{\mathtt{T\,x}}$) {return e;}	*method declarations*
N	::=	C<a>	*class types*
T, U	::=	∃$\overline{\mathtt{o}}$.N	*types*
a	::=	o | this	*contexts*
Γ	::=	$\overline{\mathtt{x{:}T}}$	*variable environments*
o, owner			*formal owners*
C, D			*classes*
f			*field names*
m			*method names*

Figure 7: Jo∃$^-$: syntax.

As example of context quantification, in Jo∃, a list with an unknown owner (and contents in the root context) can be represented as `∃o.List<o, world>`[2] and is a supertype of a list owned by any particular context (e.g., `List<this, world>`).

In this paper, we will develop a variation of Jo∃ which we call Jo∃$^-$. Our variation is mostly a subset of Jo∃, which mirrors the expressivity of Universe types and is much less expressive than Jo∃. It is simplified by removing type parameters, bounds on formal contexts, variables as contexts, parametric methods, and multiple context parameters. It does, however, support subclassing, which Jo∃ does not, and implicitly packs and unpacks existential types.

We give the syntax for Jo∃$^-$ in Fig. 7. The syntax of expressions is similar to Universes, only object creation is changed, where an owner must be supplied. Class declarations must come with a single context parameter — the owner of instantiations of the class. There is no distinction between source types and internal types in Jo∃$^-$. Types are existentially quantified class types parameterised by a single context[3]. An existential type may be quantified by the empty set, which is analogous to an unquantified type. For convenience, we use $\exists\overline{\mathtt{o}}.\mathtt{T}$ for $\exists\overline{\mathtt{o}},\overline{\mathtt{o}'}.\mathtt{N}$ where $\mathtt{T} = \exists\overline{\mathtt{o}'}.\mathtt{N}$.

We give subtyping in Fig. 8. This follows subclassing and existential subtyping; the latter is given by ∃S-Env and follows Tame FJ [9] and other models for Java wildcards. In Jo∃$^-$, ∃S-Env allows subtyping between concrete and existential types (e.g., $\vdash_\exists$ `C<this>` <: `∃o.C<o>`) and between equivalent existential types (e.g., $\vdash_\exists$ `∃o1,o2.C<o1>` <: `∃o1.C<o1>` and $\vdash_\exists$ `∃o1.C<o1>` <: `∃o1,o2.C<o1>`).

We give rules for well-formed types and contexts in Fig. 9. A type is well-formed if the class part is declared in the program and the context parameter is well-formed (taking into account any quantification). Well-formed contexts may only be quantified contexts, `this`, and the owner for the current class, because these are the only contexts put into

[2] Quantified contexts in Jo∃ should be bounded, we omit bounds here for clarity.

[3] Since types are only parameterised by a single context, at most one formal context in a quantifying environment will be relevant; however, the formalisation is more straightforward if we allow quantification by multiple formal contexts.

$$\frac{\mathtt{class\ C<o>} \lhd \mathtt{D<o>\ ...}}{\vdash_\exists \exists\overline{\mathtt{o}}.\mathtt{C<a>} <: \exists\overline{\mathtt{o}}.\mathtt{D<a>}}\ (\exists\text{S-Sub-Class})
\qquad
\frac{\begin{array}{c}\overline{\mathtt{o}'} \cap fv(\exists\overline{\mathtt{o}}.\mathtt{N}) = \emptyset \\ fv(\overline{\mathtt{a}}) \subseteq fv(\exists\overline{\mathtt{o}}.\mathtt{N}) \cup \overline{\mathtt{o}'}\end{array}}{\vdash_\exists \exists\overline{\mathtt{o}'}.[\overline{\mathtt{a/o}}]\mathtt{N} <: \exists\overline{\mathtt{o}}.\mathtt{N}}\ (\exists\text{S-Env})$$

$$\frac{}{\vdash_\exists \mathtt{T} <: \mathtt{T}}\ (\exists\text{S-Reflex})
\qquad
\frac{\begin{array}{c}\vdash_\exists \mathtt{T}_1 <: \mathtt{T}_3 \\ \vdash_\exists \mathtt{T}_3 <: \mathtt{T}_2\end{array}}{\vdash_\exists \mathtt{T}_1 <: \mathtt{T}_2}\ (\exists\text{S-Trans})$$

Figure 8: Jo$\exists^-$: subtyping.

$$\frac{\mathtt{o} \in \overline{\mathtt{o}}}{\overline{\mathtt{o}};\Gamma \vdash_\exists \mathtt{o}\ \text{OK}}\ (\exists\text{F-Context})
\qquad
\frac{\mathtt{x} \in dom(\Gamma)}{\overline{\mathtt{o}};\Gamma \vdash_\exists \mathtt{x}\ \text{OK}}\ (\exists\text{F-Var})$$

$$\frac{\begin{array}{c}\mathtt{class\ C<o>...} \\ \overline{\mathtt{o}};\Gamma \vdash_\exists \mathtt{a}\ \text{OK}\end{array}}{\overline{\mathtt{o}};\Gamma \vdash_\exists \mathtt{C<a>}\ \text{OK}}\ (\exists\text{F-Class})
\qquad
\frac{\overline{\mathtt{o}},\overline{\mathtt{o}'};\Gamma \vdash_\exists \mathtt{N}\ \text{OK}}{\overline{\mathtt{o}};\Gamma \vdash_\exists \exists\overline{\mathtt{o}'}.\mathtt{N}\ \text{OK}}\ (\exists\text{F-Exist})$$

Figure 9: Jo$\exists^-$: well-formed contexts and types.

the environments in $\exists$T-Class [7]. We keep the rules general for simplicity and to stay close to Jo$\exists$.

We give rules to type check expressions in Fig. 10; they are mostly standard. The two interesting innovations concern existential unpacking and packing (which is done implicitly, that is, without expressions, in contrast to Jo$\exists$), and using the *sv* function to assist in the handling of representation exposure. The type of the receiver ($\exists\overline{\mathtt{o}'}.\mathtt{N}$) in method calls, field accesses, and assignments is unpacked by using `N` without quantification for type lookups. The quantifying variables ($\overline{\mathtt{o}'}$) are used to quantify the assigned type (existential packing) in the conclusion of the rules.

The *sv* function is used to ensure that expressions are not substituted into types. The context `this` may appear in the declared types of fields and methods; during type checking, `this` must be substituted away. If the receiver in the expression being typed is a context (i.e., `this` in Jo$\exists^-$), then it may be used[4], otherwise we use a fresh context variable, which is then quantified in the assigned type of the expression. For example, in the list class defined at the start of this section, `pf` is declared with type `Object<this>`; if `this` has type `List<o>`, we can assign the type `Object<this>` to `this.pf` as we can do the substitution `this/this`. In checking `x.pf` (assuming `x` has type `List<o>`), we cannot do the substitution `x/this` because `x` is not a context in Jo$\exists^-$. Instead we substitute the fresh context $\mathtt{o}'$, giving `Object<o'>`; we can then assign the type $\exists\mathtt{o}.\mathtt{Object<o>}$ to `x.pf` which prevents $\mathtt{o}'$ becoming free.

This is a novel use of existential quantification and avoids us being unable to type expressions where the receiver is not a context. It is safe because the fresh context variable introduced cannot be matched to any other context by subtyping.

$$\frac{}{\overline{\mathtt{o}};\Gamma \vdash_\exists \mathtt{x} : \Gamma(\mathtt{x})}\ (\exists\text{T-Var})
\qquad
\frac{\overline{\mathtt{o}};\Gamma \vdash_\exists \mathtt{T}\ \text{OK}}{\overline{\mathtt{o}};\Gamma \vdash_\exists \mathtt{null} : \mathtt{T}}\ (\exists\text{T-Null})$$

$$\frac{\overline{\mathtt{o}};\Gamma \vdash_\exists \mathtt{C<a>}\ \text{OK}}{\overline{\mathtt{o}};\Gamma \vdash_\exists \mathtt{new\ C<a>} : \mathtt{C<a>}}\ (\exists\text{T-New})$$

$$\frac{\begin{array}{c}\overline{\mathtt{o}};\Gamma \vdash_\exists \mathtt{e} : \exists\overline{\mathtt{o}'}.\mathtt{N} \\ fType(\mathtt{f},\mathtt{N}) = \mathtt{T}\end{array}}{\overline{\mathtt{o}};\Gamma \vdash_\exists \mathtt{e.f} : \exists\overline{\mathtt{o}'},\overline{\mathtt{o}''}.[sv_{\overline{\mathtt{o}''}}(\mathtt{e})\mathtt{/this}]\mathtt{T}}\ (\exists\text{T-Field})$$

$$\frac{\begin{array}{c}\overline{\mathtt{o}};\Gamma \vdash_\exists \mathtt{e} : \exists\overline{\mathtt{o}'}.\mathtt{N} \qquad fType(\mathtt{f},\mathtt{N}) = \mathtt{T} \\ \overline{\mathtt{o}};\Gamma \vdash_\exists \mathtt{e}' : \mathtt{T}' \\ \vdash_\exists \mathtt{T}' <: [sv_{\overline{\mathtt{o}'}}(\mathtt{e})\mathtt{/this}]\mathtt{T}\end{array}}{\overline{\mathtt{o}};\Gamma \vdash_\exists \mathtt{e.f\ =\ e}' : \exists\overline{\mathtt{o}'},\overline{\mathtt{o}''}.[sv_{\overline{\mathtt{o}''}}(\mathtt{e})\mathtt{/this}]\mathtt{T}}\ (\exists\text{T-Assign})$$

$$\frac{\begin{array}{c}\overline{\mathtt{o}};\Gamma \vdash_\exists \mathtt{e} : \exists\overline{\mathtt{o}'}.\mathtt{N} \\ mType(\mathtt{m},\mathtt{N}) = \overline{\mathtt{T}}\rightarrow\mathtt{T} \qquad \overline{\mathtt{o}};\Gamma \vdash_\exists \overline{\mathtt{e} : \mathtt{U}} \\ \vdash_\exists \overline{\mathtt{U} <: [sv_{\overline{\mathtt{o}'}}(\mathtt{e})\mathtt{/this}]\mathtt{T}}\end{array}}{\overline{\mathtt{o}};\Gamma \vdash_\exists \mathtt{e.m(\overline{e})} : \exists\overline{\mathtt{o}'},\overline{\mathtt{o}''}.[sv_{\overline{\mathtt{o}''}}(\mathtt{e})\mathtt{/this}]\mathtt{T}}\ (\exists\text{T-Invk})$$

$$\frac{}{sv(\mathtt{this})_\emptyset = \mathtt{this}}
\qquad
\frac{\mathtt{e} \neq \mathtt{this}}{sv(\mathtt{e})_{\{\mathtt{o}\}} = \mathtt{o}}$$

Figure 10: Jo$\exists^-$: expression typing rules.

Therefore objects which are in another objects' representation can only be typed (and therefore referenced) abstractly [5].

We relegate method and field lookup functions and rules for type checking methods and classes to the accompanying technical report[7].

5. ENCODING UNIVERSES IN Jo$\exists^-$

There is a relatively straightforward mapping from Universe types to Jo$\exists^-$ types. We define the translation of the Universe type `T` as $[\![\mathtt{T}]\!]_{\overline{\mathtt{o}}}$. Since the translation function is not one-to-one, it has no inverse. Therefore, we must seperately define a translation from Jo$\exists^-$ types to Universe types. We define the translation of a Jo$\exists^-$ type `T` as $[\![\mathtt{T}]\!]^{\leftarrow}_{\overline{\mathtt{o}}}$. In both cases, $\overline{\mathtt{o}}$ is a sequence of free context variables in the Jo$\exists^-$ type. Both functions are defined in Fig. 11.

Both `peer` and `self` annotations denote objects in the same context in the ownership hierarchy (that of the current object's owner); therefore, they are encoded in Jo$\exists^-$ with the same types. The `self` annotation includes extra

[4] In Jo$\exists$, receivers must always be contexts so the problem is avoided.

[5] Note that $\overline{\mathtt{o}'}$ and $\overline{\mathtt{o}''}$ are unrelated. The assigned types are packed with respect to both sets of context variables, but they have different sources: $\overline{\mathtt{o}'}$ are unpacked from the type of the receiver, $\overline{\mathtt{o}''}$ are a result of substituting generated variables into the field or method type.

$$\frac{}{[\![\texttt{any C}]\!]_{\emptyset} = \exists\texttt{o.C<o>}} \qquad \frac{}{[\![\texttt{peer C}]\!]_{\emptyset} = \exists\emptyset\texttt{.C<owner>}}$$

$$\frac{}{[\![\texttt{rep C}]\!]_{\emptyset} = \exists\emptyset\texttt{.C<this>}} \qquad \frac{}{[\![\texttt{self C}]\!]_{\emptyset} = \exists\emptyset\texttt{.C<owner>}}$$

$$\frac{}{[\![\texttt{lost C}]\!]_{\{\texttt{o}\}} = \exists\emptyset\texttt{.C<o>}} \qquad \frac{}{[\![\texttt{new T}]\!]_{\emptyset} = \texttt{new } [\![\texttt{T}]\!]_{\emptyset}}$$

$$\frac{\texttt{o} \in \overline{\texttt{o}}}{[\![\exists\overline{\texttt{o}}\texttt{.C<o>}]\!]^{\leftarrow}_{\emptyset} = \texttt{any C}} \qquad \frac{\texttt{owner} \notin \overline{\texttt{o}}}{[\![\exists\overline{\texttt{o}}\texttt{.C<owner>}]\!]^{\leftarrow}_{\emptyset} = \texttt{peer C}}$$

$$\frac{}{[\![\exists\overline{\texttt{o}}\texttt{.C<this>}]\!]^{\leftarrow}_{\emptyset} = \texttt{rep C}}$$

$$\frac{\texttt{o} \neq \texttt{owner} \qquad \texttt{o} \notin \overline{\texttt{o}}}{[\![\exists\overline{\texttt{o}}\texttt{.C<o>}]\!]^{\leftarrow}_{\{\texttt{o}\}} = \texttt{lost C}} \qquad \frac{}{[\![\texttt{new T}]\!]^{\leftarrow}_{\emptyset} = \texttt{new } [\![\texttt{T}]\!]^{\leftarrow}_{\emptyset}}$$

Figure 11: Translation from Universes to Jo$\exists^-$ and Jo$\exists^-$ to Universes.

```
class C {
  peer Object f1;
  any Object f2;

  void m(any C x) {
    this.f1 = new peer Object();   //1: OK
    x.f1 = new peer Object();      //2: error
    x.f2 = new peer Object();      //3: OK
  }
}
```

Figure 12: Universes example program $\texttt{P}_1$.

```
class C<owner> {
  Object<owner> f1;
  ∃o. Object<o> f2;

  void m(∃o. C<o> x) {
    this.f1 = new Object<owner>();   //1: OK
    x.f1 = new Object<owner>();      //2: error
    x.f2 = new Object<owner>();      //3: OK
  }
}
```

Figure 13: Jo$\exists^-$ example program $\texttt{P}_2$.

information: a variable with this annotation can only contain the current object, `this`. In Jo$\exists^-$, this information is not stored in the types, but is used directly in the type rules, specifically in the function *sv* [6].

The difference between `any` and `lost` also becomes clear from the encoding: `any` is encoded as an existential type, intuitively the type represents an object owned by an unknown owner (the type system doesn't *care* about the owner); `lost` is encoded with a free owner variable, which means the owner is unknown, but some specific owner (the type system doens't *know* about the owner).

The translations are easily extended to expressions; object creation is the only expression that includes a type and so is the only expression that requires translation (also given in Fig. 11). All types for fields and methods in class bodies must be translated. Class declarations are translated from Universes by adding an `owner` parameter to the declared class and its superclass. Translating to a Universes program simply removes these context parameters. We also extend the translations of types to the translations of variable environments (Γ) in the obvious way.

5.1 Example

Consider the program $\texttt{P}_1$ using Universe types in Fig. 12 and the program $\texttt{P}_2$ using Jo$\exists^-$ types in Fig. 13. These two programs are equivalent, that is, $[\![\texttt{P}_1]\!]_{\emptyset} = \texttt{P}_2$ and $[\![\texttt{P}_2]\!]^{\leftarrow}_{\emptyset} = \texttt{P}_1$. Both describe the same topology and type checking in both systems rejects expression 2.

In $\texttt{P}_1$ the field update `x.f1` in expression 2 is forbidden, as the viewpoint adaptation `any` $\triangleright$ `peer Object` results in `lost Object` and `lost` is forbidden in the adapted field type. On the other hand, the field update `x.f2` in expression 3 is allowed, as `any` $\triangleright$ `any Object` results in `any Object` and the right-hand side is a correct subtype.

In expression 2 of $\texttt{P}_2$, the type of `x` must be unpacked before it can be used. Therefore, the field type lookup $\mathit{fType}(\texttt{f1}, \texttt{C<o1>})$ is performed, where `o1` is a fresh context variable. This lookup gives the type `Object<o1>`. There is no subtype relationship between `Object<owner>` and `Object<o1>` because their parameters do not match and subtyping of unquantified types is invariant.

In expression 3, the lookup $\mathit{fType}(\texttt{f2}, \texttt{C<o1>})$ results in $\exists\texttt{o.Object<o>}$, which is a supertype of `Object<owner>`, because of the variance of existential types, and the assignment is allowed.

5.2 Properties of the Encoding

We wish to state that the Universes and Jo$\exists^-$ type systems are equivalent; i.e., an expression will type check in one if and only if it type checks in the other. Formally,

> **Theorem — Equivalence of Universes and Jo$\exists^-$:** For all `e` , Γ holds: there exists `T` such that $\Gamma \vdash_u \texttt{e} : \texttt{T}$ if and only if there exists $\texttt{T}'$ such that $\texttt{owner}; [\![\Gamma]\!]_{\emptyset} \vdash_{\exists} [\![\texttt{e}]\!]_{\emptyset} : \texttt{T}'$

We use our alternate type system for Universes (described in Sect. 3.1) as an intermediate step between the two Jo$\exists^-$ and Universes. We have proved the following lemma:

> **Lemma — Correspondence of Universes and our Alternate Formalisation of Uni-**

[6] Alternatively, we could modify the Universes system to remove the special treatment of `this` expressions and thus remove the need for the `self` annotation, or add a `self` annotation to Jo$\exists^-$. However, as the aim of this work is to demonstrate the connection between Universes and ownership using standard type-theoretic features, neither situation is ideal: the first muddies the definition of Universes, the second adds a non-standard element.

verses: For all $\mathtt{e}$, $\mathtt{T}$, Γ holds: $\Gamma \vdash_u \mathtt{e} : \mathtt{T}$ if and only if $\Gamma \vdash_{u'} \mathtt{e} : \mathit{close}(\mathtt{T})$.

The fact that we assign a different type to $\mathtt{e}$ in the two systems is not important because in the main theorem, we do not claim a correspondence between $\mathtt{T}$ and $\mathtt{T}'$[7].

The above lemma requires the following lemma concerning subtyping in the two systems:

Lemma — Correspondence of Universes Subtyping and Subtyping in our Alternate Formalisation of Universes: For all $\mathtt{T}$, $\mathtt{T}'$ holds: $\vdash_u \mathtt{T} <: \mathtt{T}'$ and $\mathtt{T}' \neq \mathtt{lost}\ \_$ if and only if $\vdash_{u'} \mathtt{T} <: \mathtt{T}'$.

Proving this lemma requires the use of transitivity-free subtyping for Universes, given in the accompanying technical report[7].

In order to prove the final stage of the equivalence, that alternate Universe subtyping corresponds with Jo$\exists^-$ subtyping, we require that subtyping in these two systems corresponds. This lemma requires the use of transitivity-free subtyping for Jo$\exists^-$, also in [7]. Due to the non-invertability of the translation, we must state this lemma in two parts:

Lemma — Correspondence of Subtyping in our Alternate Formalisation of Universes and Jo$\exists^-$ Subtyping, a: For all $\mathtt{T}$, $\mathtt{T}'$, if $\vdash_{u'} \mathtt{T} <: \mathtt{T}'$ then $\vdash_\exists \llbracket \mathtt{T} \rrbracket_{\overline{\circ}} <: \llbracket \mathtt{T}' \rrbracket_\emptyset$.

Lemma — Correspondence of Subtyping in our Alternate Formalisation of Universes and Jo$\exists^-$ Subtyping, b: For all $\mathtt{T}$, $\mathtt{T}'$, if $\vdash_\exists \mathtt{T} <: \mathtt{T}'$ then $\vdash_{u'} \llbracket \mathtt{T} \rrbracket_{\overline{\circ}}^{\leftarrow} <: \llbracket \mathtt{T}' \rrbracket_\emptyset^{\leftarrow}$.

In these lemmas, we only allow free context variables in the subtype ($\mathtt{T}$, not $\mathtt{T}'$); this is indicated by the subscript free variable lists of the translation functions. This corresponds to subtyping in our alternate Universes formalisation, where `lost` (which indicates a free variable) can never appear on the right-hand side of a subtype relation.

Likewise, the correspondence between typing expressions in the two languages must be stated in two parts:

Lemma — Correspondence of our Alternate Formalisation of Universes and Jo$\exists^-$, a: For all $\mathtt{e}$, $\mathtt{T}$, Γ, if $\Gamma \vdash_{u'} \mathtt{e} : \mathtt{T}$ then $\mathtt{owner}; \llbracket \Gamma \rrbracket_\emptyset \vdash_\exists \llbracket \mathtt{e} \rrbracket_\emptyset : \llbracket \mathtt{T} \rrbracket_\emptyset$.

Lemma — Correspondence of our Alternate Formalisation of Universes and Jo$\exists^-$, b: For all $\mathtt{e}$, $\mathtt{T}$, Γ, if $\mathtt{owner}; \Gamma \vdash_\exists \mathtt{e} : \mathtt{T}$ then $\llbracket \Gamma \rrbracket_\emptyset^{\leftarrow} \vdash_{u'} \llbracket \mathtt{e} \rrbracket_\emptyset^{\leftarrow} : \llbracket \mathtt{T} \rrbracket_\emptyset^{\leftarrow}$.

In these lemmas, the translation function may not create free variables, indicated by the subscript $\emptyset$. This means we do not attempt to show a correspondence where there are free variables in the Jo$\exists^-$ types; this is a basic well-formedness property of the types.

In summary, we have proved that:

Lemma — Correspondence of Universes and Jo$\exists^-$, a: For all $\mathtt{e}$, $\mathtt{T}$, Γ, if $\Gamma \vdash_u \mathtt{e} : \mathtt{T}$ then $\mathtt{owner}; \llbracket \Gamma \rrbracket_\emptyset \vdash_\exists \llbracket \mathtt{e} \rrbracket_\emptyset : \llbracket \mathit{close}(\mathtt{T}) \rrbracket_\emptyset$.

Lemma — Correspondence of Universes and Jo$\exists^-$, b: For all $\mathtt{e}$, $\mathtt{T}$, Γ, if $\mathtt{owner}; \Gamma \vdash_\exists \mathtt{e} : \mathtt{T}$ then $\llbracket \Gamma \rrbracket_\emptyset^{\leftarrow} \vdash_u \llbracket \mathtt{e} \rrbracket_\emptyset^{\leftarrow} : \mathtt{U}$ where $\mathit{close}(\mathtt{U}) = \llbracket \mathtt{T} \rrbracket_\emptyset^{\leftarrow}$.

Both these results follow from the above lemmas, and together give the main result of this section.

Full proofs can be downloaded from: http://www.doc.ic.ac.uk/~ncameron/papers/cameron_iwaco09_proofs.pdf

6. CONCLUSION

We have formalised an encoding from Universe types to parametric ownership types with existential quantification. We have also defined the reverse encoding, and shown that both are sound. Up to the non-invertability of the encodings, this also gives completeness for both encodings. Essentially, we have shown that both systems describe the same heap topologies. This follows from the equivalences of types and type checking.

We have expanded the understanding of the two systems' relationship and shown exactly what the Universe modifiers mean in terms of the more fundamental type theoretic tools of parameterisation and quantification.

In practical terms, the two systems have different advantages and potential markets. Universe annotations are much simpler than parametric ownership types, and are much more usable by programmers; therefore, they are more likely to be adopted in a real language. Contrariwise, parametric ownership types are more expressive and can describe the ownership hierarchy in greater detail; therefore, they are more useful for specifying internal representations of programs, or in applications where the payoff from using ownership justifies the higher annotation overhead. In addition, the systems enforce different encapsulation properties with different target applications.

Future Work.

We would like to directly prove Jo$\exists^-$ sound, even though this is a very simple variation of the proof for Jo$\exists$. We would like to extend our encoding to cover type parameters, since both systems can include this feature [16, 8].

We are investigating the structure of the heap in both systems by examining their operational semantics. Our aim is to prove that the heap topologies given by both systems are identical. Although this follows from our results and each system's soundness results, it would be interesting to formalise and prove these properties directly.

We are applying the insight gained from this work to develop a hybrid language which allows a mix-and-match combination of simple Universe annotations and parametric and path-dependent ownership types for more fine-grained specifications. This will allow programmers to easily choose the simplest and most concise specification for a particular ownership relation.

Finally, we are investigating the relationship between the encapsulation properties of these and other ownership systems.

Acknowledgements.

We would like to thank the anonymous reviewers for their useful comments. The first author's work was funded in part by a Build IT Postdoctoral fellowship.

[7] In fact such a correspondence exists, based on *close*.

7. REFERENCES

[1] Jonathan Aldrich. *Using types to enforce architectural structure*. PhD thesis, University of Washington, 2003.

[2] Chris Andreae, Yvonne Coady, Celina Gibbs, James Noble, Jan Vitek, and Tian Zhao. Scoped types and aspects for real-time systems. In *European Conference on Object Oriented Programming (ECOOP)*, 2006.

[3] Anindya Banerjee and David Naumann. Ownership confinement ensures representation independence for object-oriented programs. *JACM: Journal of the ACM*, 2005.

[4] Chandrasekhar Boyapati, Robert Lee, and Martin C. Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2002.

[5] Chandrasekhar Boyapati, Alexandru Salcianu, William S. Beebee, and Martin C. Rinard. Ownership types for safe region-based memory management in real-time java. In *Programming Language Design and Implementation (PLDI)*, 2003.

[6] Nicholas Cameron. *Existential Types for Variance — Java Wildcards and Ownership Types*. PhD thesis, Imperial College London, 2009.

[7] Nicholas Cameron and Werner Dietl. Comparing Universes and Existential Ownership Types. Technical Report 06, School of Engineering and Computer Science, VUW, 2009. https://ecs.victoria.ac.nz/twiki/pub/Main/TechnicalReportSeries/ECSTR09-06.pdf.

[8] Nicholas Cameron and Sophia Drossopoulou. Existential Quantification for Variant Ownership. In *European Symposium on Programming Languages and Systems (ESOP)*, 2009.

[9] Nicholas Cameron, Sophia Drossopoulou, and Erik Ernst. A Model for Java with Wildcards. In *European Conference on Object Oriented Programming (ECOOP)*, 2008.

[10] Nicholas Cameron, Sophia Drossopoulou, James Noble, and Matthew Smith. Multiple Ownership. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2007.

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

[12] David G. Clarke and Sophia Drossopoulou. Ownership, Encapsulation and the Disjointness of Type and Effect. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2002.

[13] David G. Clarke, John M. Potter, and James Noble. Ownership Types for Flexible Alias Protection. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1998.

[14] David Cunningham, Werner Dietl, Sophia Drossopoulou, Adrian Francalanza, Peter Müller, and Alexander J. Summers. Universe Types for Topology and Encapsulation. In *Formal Methods for Components and Objects (FMCO)*, 2008.

[15] David Cunningham, Sophia Drossopoulou, and Susan Eisenbach. Universe Types for Race Safety. In *Verification and Analysis of Multi-threaded Java-like Programs (VAMP)*, 2007.

[16] Werner Dietl, Sophia Drossopoulou, and Peter Müller. Generic Universe Types. In *European Conference on Object Oriented Programming (ECOOP)*, 2007.

[17] Werner Dietl and Peter Müller. Universes: Lightweight Ownership for JML. *Journal of Object Technology*, 4(8):5–32, 2005.

[18] Werner Dietl and Peter Müller. Ownership type systems and dependent classes. In *Foundations of Object-Oriented Languages (FOOL)*, 2008.

[19] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a Minimal Core Calculus For Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001. An earlier version of this work appeared at OOPSLA'99.

[20] Bart Jacobs, Frank Piessens, K. Rustan M. Leino, and Wolfram Schulte. Safe concurrency for aggregate objects with invariants. In *Software Engineering and Formal Methods (SEFM)*, 2005.

[21] Neel Krishnaswami and Jonathan Aldrich. Permission-Based Ownership: Encapsulating State in Higher-Order Typed Languages. In *Programming Language Design and Implementation (PLDI)*, 2005.

[22] K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts. In *European Conference on Object-Oriented Programming (ECOOP)*, 2004.

[23] Yi Lu and John Potter. On Ownership and Accessibility. In *European Conference on Object Oriented Programming (ECOOP)*, 2006.

[24] Yi Lu and John Potter. Protecting Representation with Effect Encapsulation. In *Principles of Programming Languages (POPL)*, 2006.

[25] Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[26] Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular Invariants for Layered Object Structures. *Science of Computer Programming*, 62(3):253–286, October 2006.

[27] James Noble, Jan Vitek, and John Potter. Flexible Alias Protection. In *European Conference on Object Oriented Programming (ECOOP)*, 1998.

[28] Mads Torgersen, Christian Plesner Hansen, Erik Ernst, Peter von der Ahé, Gilad Bracha, and Neal Gafter. Adding Wildcards to the Java Programming Language. *Journal of Object Technology*, 3(11):97–116, 2004. Special issue: OOPS track at SAC 2004, Nicosia/Cyprus.

[29] Tobias Wrigstad and David G. Clarke. Existential Owners for Ownership Types. *Journal of Object Technology*, 6(4), 2007.