Tribal Ownership

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

James Noble
Victoria University of Wellington
kjx@ecs.vuw.ac.nz

Tobias Wrigstad
Uppsala University
tobias.wrigstad@it.uu.se

Abstract

Tribal Ownership unifies class nesting and object ownership. Tribal Ownership is based on Tribe, a language with nested classes and object families. In Tribal Ownership, a program's runtime object ownership structure is characterised by the lexical nesting structure of its classes.

We build on a variant of Tribe to present a *descriptive* ownership system, using object nesting to describe heap partitions, but without imposing any restrictions on programming disciplines. We then demonstrate how a range of different *prescriptive ownership policies* can be supported on top of the descriptive Tribal Ownership mechanism; including a novel *owners-as-local-dominators* policy. We formalise our type system and prove soundness and several ownership invariants. The resulting system requires strikingly few annotations, and uses well-understood encapsulation techniques to create ownership systems that should be intuitive for programmers.

Categories and Subject Descriptors D.3.3 [*Software*]: Programming Languages—Language Constructs and Features

General Terms Languages, Theory

Keywords Ownership types, virtual classes, nested classes, family polymorphism

1. Introduction

Ownership Types Ownership type systems statically impose hierarchical structures onto the heap. By structuring the heap, compilers and programmers can reason about small parts of it in isolation, which makes many otherwise intractable analyses possible. Ownership languages often support encapsulation policies based on the heap structure (*prescriptive* systems); these might restrict references, modification, or access. Alternatively, ownership information can be used to describe the effects of computation without imposing any restrictions (*descriptive* ownership) [16]. Ownership types have been used to support memory management and garbage collection [10, 51], alias analysis [2], verification [25], concurrency [7, 24, 50], parallelisation [21, 18], real-time programming [46, 51], and more [2, 23, 9].

Ownership types tend to require substantial annotations to describe how individual fields and methods relate to the ownership hierarchy. These annotations can be confusing, and have negative side-effects on refactoring and program maintenance. Furthermore, because the annotations primarily decorate individual methods and fields, it can be difficult to see how each class or object fits into the ownership structure as a whole.

In this paper we present an ownership system based on virtual classes that does not require any additional annotation, and, by making the ownership structure explicit in a program's class structure, should be easier for programmers to use and understand.

Virtual Classes Virtual classes allow *families* of classes to be inherited, rather than just single classes. In a language with virtual classes, classes are lexically nested inside other classes. When a class is inherited, its nested inner classes (and the relationships between them) are inherited along with its methods and fields. Virtual classes allow programs to express types such as "all nodes of any graph", "all nodes belonging to the graph g", and "this particular node in this particular graph", and to distinguish between them statically.

Virtual classes are a powerful and intuitive mechanism which are beginning to appear in practical programming languages [36, 12]. We expect virtual classes to become more mainstream; our proposal uses virtual classes to bring the considerable benefits of ownership types to a language at low additional cost.

Our Contribution In this paper we describe *Tribal ownership*, an ownership types system built on a variant of the virtual class calculus Tribe [19]. Our language has no additional syntax to support ownership, only the usual syntax used to support virtual classes. In Tribal Ownership, the object ownership and nested class hierarchies coincide. By unifying these hierarchies we make the ownership hierarchy

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
OOPSLA/SPLASH'10, October 17–21, 2010, Reno/Tahoe, Nevada, USA.
Copyright © 2010 ACM 978-1-4503-0203-6/10/10. . . $10.00

more apparent in source code and the system easier for programmers to understand.

In our proposed system, all instances of a class are owned by the *object* which encloses that class. Since this object is already tracked by the type system, no additional annotations are required to enforce ownership. Flexibility is provided by allowing classes to be *imported* into a different class, and so take instances of the importing class as owners.

We show how different encapsulation policies can be enforced on top of the descriptive Tribal Ownership system. Additionally, we present a new variation on the standard owners-as-dominators policy that enforces owners-as-dominators locally within isolated sub-heaps, whilst giving full access to other areas of the heap.

In summary, the contributions of this paper are:

- A new variant of the Tribe language with simpler types, cross-family inheritance, and generics. We believe this variant is easier to program with and understand than the original version of Tribe. We also contribute a new formalism (in which, formalising cross-family inheritance was the main challenge) and type soundness proof.
- The first formalisation of ownership types using virtual classes (Tribe$_{own}$), and a proof that the ownership hierarchy is preserved by our type system.
- Formal and informal descriptions of several different encapsulation policies built on top of Tribal ownership.
- The novel encapsulation policy owners-as-*local*-dominators, a flexible policy which subsumes owners-as-dominators. We prove that this property is enforced by our formalism.

Organisation We give background on the Tribe language in Sect. 2. We describe our new variant, Tribe$_{own}$, its formalisation, and soundness proof in Sect. 3. We describe how this system supports descriptive ownership, and extend our formalism in order to prove sound ownership typing in Sect. 4. We describe how encapsulation policies can be encoded in Tribe$_{own}$ and describe and formalise our new encapsulation property owners-as-local-dominators in Sect. 5. We cover related work in Sect. 6, and further work and our conclusions in Sect. 7.

2. Tribe

Tribe (along with many other languages following on from BETA [37]) supports *nested virtual classes*. As with Java's (non-static) inner classes, Tribe's nested class declarations lead to nested objects at runtime. Consider the following simple example, inspired by [40]:

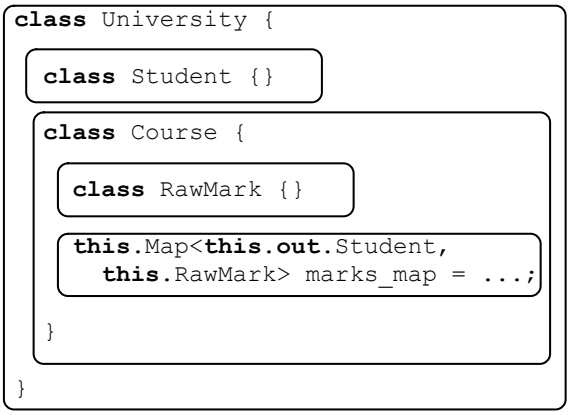

Figure 1. Nesting structure of nested classes

```
class University {
  class Student {...}
  class Course {
    class RawMark {...}
    this.Map<this.out.Student,
      this.RawMark> marks_map = ...;
  }
}
```

This code defines a `University` class with two nested classes inside it: one class for modelling `Students`, and another for modelling `Courses`. Each `Course` object has a `RawMark` class, representing the private, unmoderated marks for each course, and a `Map` which relates students to their raw marks. Figure 1 shows how the lexical structure of the class definitions imposes a nesting relationship on the classes.

The `marks_map` field maps students (`this.out.Student`, i.e., student objects from the enclosing university object) to raw marks (`this.RawMark`). The respective types reflect that `Student` is defined one level out in the nesting hierarchy, and `RawMark` is defined inside the current class (`Course`).

2.1 Nested Objects

Fig. 2 shows an example of how these nested classes could be instantiated at runtime. The outermost `University` class is instantiated twice, with one instance representing "VUW" and the other representing "Uppsala". Nested inside each university *object* are a number of different objects, instances of the `Student` and `Course` classes. Just as the `Student` and `Course` class declarations are nested inside the declaration of the `University` class in the program source code, so the various `Student` and `Course` instances are nested inside the `University` object to which belong. Going further down inside the `VUW` course `SWEN221`, inside that course object are `Map` and three `RawMark` objects. Being nested inside another object has no effect on references between objects: the `Map` refers to those three `RawMarks` and also to three students who

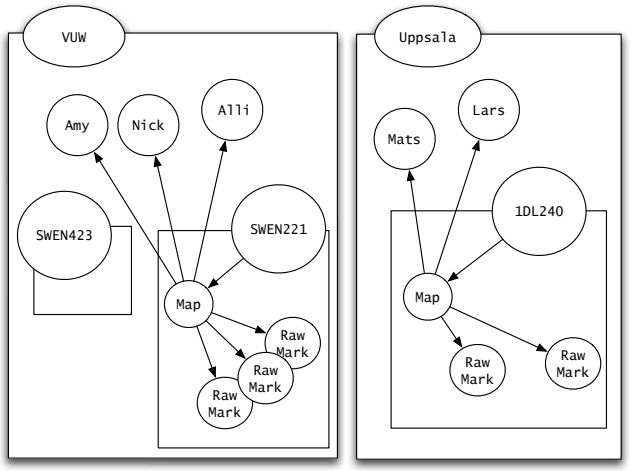

Figure 2. Objects instantiating nested classes

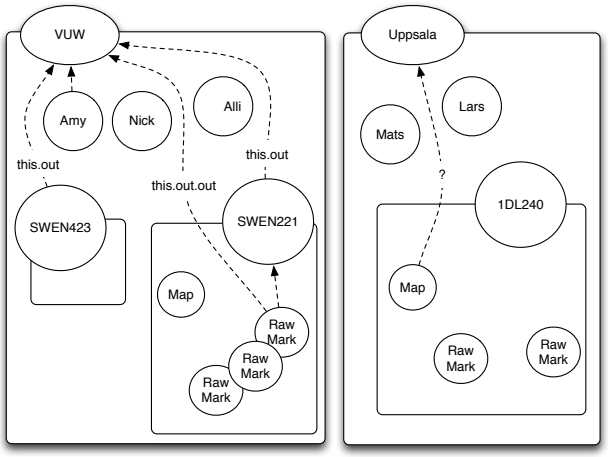

Figure 3. Paths from nested to enclosing objects.

are presumably taking those courses (and have the associated raw marks) but who are outside the `Course` object.

In Tribe, inner class instances maintain a runtime reference to their enclosing object: this is written "**`out`**". So **`out`** for `Mats`, `Lars`, and `1DL240` is the `Uppsala` object; while **`out`** for `Amy`, `Nick`, `Alli`, `SWEN423`, and `SWEN211` is the `VUW` object. This is illustrated in Fig. 3. Note the ? on the dotted line from `Map` to `Uppsala`, which we will return to when discussing cross-family imports later.

The **`out`** above is the same **`out`** used in defining types: conceptually, the *types* of the inner classes are also nested within their enclosing object. Every object anchors a *class family*: objects in the class are only compatible (related by subtyping) with objects in the same family. So the courses and students in each university are *different* types from *different* families: a VUW student cannot enroll at an Uppsala course, and vice versa [29, 19].

2.2 Family Polymorphism

Tribe supports a powerful notion of inheritance which allows groups of classes to be extended together, preserving the relationships between classes. Nested classes can be inherited or overridden along with a superclass's fields and methods. This is known as *family polymorphism* [29], and supports sophisticated implementations for design patterns and the expression problem [48, 47]. The following example shows how a `Conservatorium` — a stand-alone music school — can be implemented by inheriting from a `University`:

```
class Conservatorium extends University {
  class Student {this.out.Instrument major; ...}
}

class Instrument {...}
```

Because they include nested inner classes, `University` and `Conservatorium` define two class *families*, which are related by inheritance. So, `Conservatorium` will not only inherit methods and fields from `University`, but also `University`'s `Course` and `Student` classes. Because every conservatorium student must major in a musical instrument, `Conservatorium.Student` *further binds* `University.Student`: adding in a `major` field of type `Instrument`. This is called further binding because by class family inheritance, a `Conservatorium.Student` (i.e. the `Student` class declared within `Conservatorium`) will also gain all the fields and methods from the `University.Student` student — and any inner classes too!

Subtyping in Tribe follows from subclassing, but not further binding. For example, `Conservatorium` is a subtype of `University`; but `Conservatorium.Student` is not a subtype of `University.Student`

2.3 Tribe Types

Tribe uses *path dependent types* to distinguish classes which belong to different families. For example, assuming the variables `University u` and `Conservatorium c`, the type `c.Course` is not a subtype of `u.Course`, because `u` and `c` might refer to different objects. *Object family types* consist of a *path* to an object and the name of a class defined by that object. Paths start at a variable (often **`this`**) and have steps using final fields or the **`out`** keyword (which denotes a class's enclosing object). For example, in an object with type `u.Course`, the path **`this.out`**, will refer to a `University` object, in an object of type `c.Course`, the same path will refer to a `Conservatorium` object.

Tribe types form a spectrum: from *singleton types* which consist of a path only and denote a specific object; object family types which represent instantiations of a class nested in a specific object; to *absolute types* which specify only the class. In our running example, `u` is a singleton type denoting the specific university object referred by the variable; `u.Course` is an object family type denoting any course in `u`, and the absolute type `University.Course` is any course in any university.

Subtyping only occurs within an object family: `u.Course` is a subtype of `u.Course` (by reflexivity), but `u1.Course` is not a subtype of `u2.Course`; all three types are subtypes of their corresponding absolute type, `University.Course` (assuming `u`, `u1`, `u2` are instances of `University`).

Tribe is not only expressive, but *intuitive*: nested classes are easy to understand, families of classes are a natural unit of code, and paths are familiar from Java programming.

3. Tribe$_{own}$

Our goal has been to support ownership types in a virtual classes language. We chose Tribe as our base language because of its flexible types, elegant and fairly standard formalism, and because the type system is amenable to capturing ownership information. Supporting object ownership in Tribe is not trivial, however: we had to improve support for cross-family inheritance (described below in Sect. 3.2), change some aspects of the formalisation (Sect. 3.3) and add ways to reason about and enforce ownership properties (Sect. 4 and Sect. 5). We call our variant language Tribe$_{own}$.

In Tribe$_{own}$, we take the hierarchy of nested classes as the ownership hierarchy. An object's owner is its enclosing object. For any type `T`, `T.out` (the type of the enclosing object) is the owner of objects with type `T`.

3.1 Path Types and generics

Tribe$_{own}$ has a more restrictive syntax of types than Tribe: we disallow the use of final field names in types and restrict the ordering of the components of types, in effect, imposing a normal form for types. This is necessary to conceptualise and prove some ownership properties (discussed in Sect. 4). It also simplifies our formalisation.

Tribe types have syntax `x.(ff | out | C)*`, where `x` includes **`this`** and the outermost 'world' object, **`world`**, `ff` is the name of a final field, and `C` is a class name. Tribe$_{own}$ types have the form `x.out*.C*`; that is, a path which starts at a variable, then goes outward some number of steps, and then inward by some number of class names. We support absolute types (where the path starts at **`world`** and has no outward steps), singleton types (where there are no class names), and all types with an intermediate level of precision (e.g., **`this.out`**`.C.D`). We do not support types with fields[1] or types which have an outward step from a class name. Types of the latter case can be represented in Tribe$_{own}$ by using the equivalent type, which has neither class name nor outward step, e.g., `x.D` for `x.C.out.D`.

To make up some of the expressivity lost by removing fields from types, we introduce simple type genericity. For now, we only support unbounded generics on classes and types. Generic methods should be easy to support and are elided. Bounded and variant generics are left for future work.

[1] We found field types uncommon in our experience, and often restrictive because the Tribe type system cannot associate fields used in types with the values instantiating the fields; e.g., even if `x.f = y`, there is no relationship between `y.C` and `x.f.C`.

3.2 Importing Module Classes

Tribe$_{own}$ supports cross-family inheritance through *module classes*. Module classes restrict external dependencies by forbidding the use of **`out`** in their definitions. Since module classes do not have any external dependencies, they (and their nested classes) can be imported arbitrarily. As the code example below shows, `Map` is a module class, and therefore cannot name any external object. Hence, the dotted-line from `Map` to `Uppsala` in Fig. 3, cannot exist.

Module classes may only extend other module classes. We elaborate the example from Sect. 2:

```
module Collections {
    class List<X> {...}
    class Map<K, V> {...}
}

class University {
  ...
  class Course imports Collections{
    ...
    this.Map<this.out.Student,
      this.RawMark> marks_map = ...;
  }
}
```

Importing a module gives subtyping: in the example above, `Course` is a subtype of `Collections`. `Course.List<University>` is a subtype of `Collections.List<University>`. Module classes are reminiscent of mixin modules, which lack a super class, except that the dependency is on nesting rather than on subclassing.

In our formalism (described below), **`out`** is modelled as a distinguished field. A module class is simply a class which does not define an **`out`** field. Thus, we check whether an **`out`** field is defined or not to check if a class is a module, and can thus be imported.

Cross-family inheritance via module classes allows Tribe$_{own}$ to have a single top type. Tribe lacks a top type — each class has a nested `Object` class which is unrelated to other `Object` classes. We assume a module class `Root` nested directly inside **`world`** which contains an `Object` class. All classes (including **`world`**) implicitly import `Root`, and all classes without explicit superclasses are implicit subclasses of the adopted `Object` (a 'local' top type). Every local `Object` is a subtype of `Root.Object`, which is thus a top type for Tribe$_{own}$. We can, therefore, describe any object nested within an object or class using a local `Object` (e.g., `x.Object` is the superclass of all classes nested in `x`) and describe all objects using `Root.Object`. A top type is useful for describing variables to be used as owners and, hypothetically, for bounds on type variables.

Tribe did not support cross-family polymorphism, although an *adoption* mechanism was proposed (but not

proved sound) which addressed some of the same motivations. Module classes are a more developed solution: we formalise their interaction with subclassing and subtyping, and include them in our soundness proof.

3.3 Formalisation

Our formalisation of Tribe$_{own}$ borrows heavily from the Tribe formalism. Differences reflect differences in the language design, some simplifications, a focus on ownership rather then general language design, and differences in style. Our formalism is in the Featherweight Java [32] mould.

Languages with multiple inheritance (including Tribe$_{own}$ and Tribe) must deal with the possibility of ambiguous lookup, where fields and methods with the same name are inherited from several different superclasses. Tribe sidesteps this problem by non-deterministically selecting one. This is not an issue for soundness because all methods inherited by a class with the same name, must have the same type. We go a step further and do not even guarantee that a method from the most specific class will be executed, only that some inherited method will be executed. Although this decision would be impractical in a real language, it does not affect our soundness result because our (non-deterministic) calculus is more general than a corresponding language with deterministic dispatch. In a practical implementation, we can select from a variety of solutions, e.g., forcing the programmer to manually resolve ambiguities.

Syntax The syntax of Tribe$_{own}$ is defined in Fig. 4. Elements in grey may appear during evaluation, but may not be written by a programmer. The syntax of expressions will be described when we cover their type rules. The key feature of class declarations is that they may include nested classes. We use $\lhd$ for "**extends**" and $\bigtriangledown$ for "**imports**". The former indicates subclassing, the latter importing a module. We discuss public and private classes in Sect. 5. We treat **this** as a distinguished variable and **out** as a distinguished field.

Environments (Γ) can contain mappings from variables to their types, type variables (showing which type variables are in scope), and a heap ($\mathcal{H}^\Gamma$, used as an environment for type checking runtime expressions).

Our rules make use of substitution. Because of the flexibility of types in Tribe, any type can be substituted for a variable in a type and give a syntactically valid type. This is not the case in Tribe$_{own}$; for example, `[y.C/x]`^{std}`x.out.D` gives `y.C.out.D` which is not a Tribe$_{own}$ type (it is valid in Tribe). To address this, we define substitution in a non-standard way in Fig. 5. In this figure we use `[...]`^{std} to mean standard substitution. Tribe$_{own}$ substitution is thus defined to normalise ill-formed types into Tribe$_{own}$ types by matching and eliminating **out**s with class names. For example, `[y.C/x]x.out.D` is defined to give `y.D` in Tribe$_{own}$, and `[y.out.C/x]x.out.D` to give `y.out.D`.

Inheritance Tribe$_{own}$ has two forms of inheritance: subclassing and further binding. These relations (between absolute types) are defined in Fig. 6. The auxiliary function $\mathcal{P}$(defined in Fig. 9) looks up the definition of a class in the program, taking into account type substitution. Sub-classing reflects the relationship between classes denoted by the programmer using the `extends` keyword (SC-SUBCLASS). When a class is inherited, the subclass relationships between its child classes are also inherited (SC-NEST). (SC-IMPORT) accounts for subclassing due to imported modules. Further binding is the implicit inheritance of child classes when a surrounding class is inherited (FB-NEST). For concreteness, `B.C` further binds `A.C` and subclasses `B.D` in the following:

```
class A {
  class C {}
}
class B extends A {
  class D {}
  class C extends D {}
}
```

e	::=	`null` $\mid p \mid \gamma$`.f` $\mid \gamma$`.f =` e	*expressions*
		$\mid$ **let** `x:`T`=` e `in` e $\mid \gamma$`.m(`$\overline{\gamma}$`)`	
		$\mid$ **new** γ`.N` $\mid$ **err**	
v	::=	`null` $\mid$ **world** $\mid \iota$	*values*
$\mathcal{P}$	::=	$\overline{Q}\ e$	*programs*
Q	::=	α `class C<`$\overline{\texttt{X}}$`>` $\lhd \overline{\texttt{N}} \bigtriangledown \overline{A}\ \{\overline{Q}\ \overline{T\ \texttt{f;}}\ \overline{M}\}$	*class declarations*
α	::=	`public` $\mid$ `private`	*naming modifiers*
M	::=	T`m(`$\overline{T\ \texttt{x}}$`)` `{return` e`;}`	*method declarations*
s	::=	γ $\mid$ **world**	*path starts*
p	::=	$s \mid p$`.out`	*paths*
N	::=	`C<`$\overline{T}$`>`	*instantiated class names*
P	::=	$p \mid P.N$	*path types*
T, U	::=	P $\mid$ `X`	*types*
A	::=	**world** $\mid A.N$	*absolute types*
r	::=	**world** $\mid \iota$	*runtime paths*
R	::=	r`.C<`$\overline{R}$`>`	*runtime types*
Γ	::=	$\emptyset \mid \Gamma, \gamma$`:`$T \mid \Gamma$`,X` $\mid \mathcal{H}^\Gamma$	*environments*
γ	::=	`x` $\mid \iota$	*variables or addresses*
$\mathcal{H}$	::=	$\emptyset \mid \mathcal{H}, \iota \rightarrow \{R; \overline{\texttt{f}\rightarrow v}\}$	*heaps*
`x, this`			*variables*
`X`			*type variables*
`C, D, Object`			*class names*
`f`,**out**			*fields*

Figure 4. Syntax of Tribe$_{own}$.

$$\frac{[T/\gamma]^{std}T' = p.\overline{\texttt{C}}.\overline{\texttt{out}}.\overline{\texttt{C}'} \qquad |\overline{\texttt{C}}| = |\overline{\texttt{out}}|}{[T/\gamma]T' = p.\overline{\texttt{C}'}}$$

Figure 5. Type subsitution in Tribe$_{own}$.

Subclassing $\boxed{\vdash A \sqsubset_s A}$

$$\frac{\alpha \texttt{ class C<}\overline{\texttt{X}}\texttt{>} \lhd \overline{N} \ ... \in \mathcal{P}(A)}{\vdash A.\texttt{C<}\overline{T}\texttt{>} \sqsubset_s A.\texttt{[}\overline{T/\texttt{X}}\texttt{]}N_i} \quad \text{(SC-SubClass)}$$

$$\frac{\mathcal{P}(A) \textit{ defined} \quad \vdash A \sqsubset_i A' \qquad \vdash A'.N \sqsubset_s A'.N'}{\vdash A.N \sqsubset_s A.N'} \quad \text{(SC-Nest)}$$

$$\frac{\alpha \texttt{ class C<}\overline{\texttt{X}}\texttt{>...}\triangledown \ \overline{A} \ ... \in \mathcal{P}(A)}{\vdash A.\texttt{C<}\overline{T}\texttt{>} \sqsubset_s \texttt{[}\overline{T/\texttt{X}}\texttt{]}A_i} \quad \text{(SC-Import)}$$

Further Binding $\boxed{\vdash A \sqsubset_f A}$

$$\frac{\vdash A \sqsubset_i A' \qquad \alpha \texttt{ class C<}\overline{\texttt{X}}\texttt{>} \ ... \in \mathcal{P}(A')}{\vdash A.\texttt{C<}\overline{T}\texttt{>} \sqsubset_f A'.\texttt{C<}\overline{T}\texttt{>}} \quad \text{(FB-Nest)}$$

Inheritance $\boxed{\vdash A \sqsubset_i A}$

$$\frac{\vdash A \sqsubset_s A'}{\vdash A \sqsubset_i A'} \quad \text{(I-SC)} \qquad \frac{\vdash A \sqsubset_f A'}{\vdash A \sqsubset_i A'} \quad \text{(I-FB)}$$

Figure 6. Tribe$_{own}$ inheritance and subclassing.

Well-formed and absolute types An absolute type (A) describes an object by a path from the **world** program root, down through the class tree, for example, **world**.C.D. The judgement $\Gamma \vdash P \Uparrow A$ holds if A is an absolute type for all objects of type P. A non-variable type in Tribe$_{own}$ is well-formed if a corresponding absolute type can be derived. Absolute types and well-formed types are defined in Fig. 8[2]. The rules that define absolute types are a combination of standard well-formedness rules for languages such as FGJ [32], and an inductive navigation of the nested class hierarchy[3].

Well-formed heaps are defined in the same figure. A heap is well-formed if each object in it has a well-formed runtime type, a sensible **out** field, and all other fields are well-typed.

Subtyping Subtyping rules are given in Fig. 7. Subtyping follows from subclassing and importing ((S-SubClass) and (S-Import)), and the expected semantics of **out** (S-Out-2).

[2] The *fType auxiliary function looks up the type of a field and is defined in Fig. 9.*

[3] Compared to Tribe, we require an inheritance premise in (A-Class) (as well as (SC-Nest)) because we do not have a *cls* judgement which takes inheritance into account.

Subtyping $\boxed{\Gamma \vdash T <: T}$

$$\frac{}{\Gamma \vdash T <: T} \quad \text{(S-Reflex)} \qquad \frac{\Gamma \vdash T_1 <: T_3 \quad \Gamma \vdash T_3 <: T_2}{\Gamma \vdash T_1 <: T_2} \quad \text{(S-Trans)}$$

$$\frac{\Gamma \vdash P \Uparrow A \quad \vdash A.N \sqsubset_s A.N'}{\Gamma \vdash P.N <: P.N'} \quad \text{(S-SubClass)} \qquad \frac{\Gamma \vdash P <: P' \quad \Gamma \vdash P'.N \text{ ok}}{\Gamma \vdash P.N <: P'.N} \quad \text{(S-Nest-Class)}$$

$$\frac{\Gamma(\gamma) = P}{\Gamma \vdash \gamma <: P} \quad \text{(S-Var)} \qquad \frac{\Gamma \vdash p \Uparrow A.N \quad \textit{fType}(\textbf{out}, A.N) \textit{ defined}}{\Gamma \vdash p <: p.\textbf{out}.N} \quad \text{(S-Out-2)}$$

$$\frac{\Gamma \vdash P <: P' \quad \Gamma \vdash P \Uparrow A \qquad \Gamma \vdash P' \Uparrow A' \quad \textit{fType}(\textbf{out}, A') \textit{ defined}}{\Gamma \vdash \texttt{[}P/\texttt{x}\texttt{]x.out} <: \texttt{[}P'/\texttt{x}\texttt{]x.out}} \quad \text{(S-Out-Nest)}$$

$$\frac{\Gamma \vdash P \Uparrow A' \qquad \vdash A' \sqsubseteq_i^* A.\texttt{C<}\overline{T}\texttt{>} \quad \alpha \texttt{ class C<}\overline{\texttt{X}}\texttt{>...}\triangledown \ \overline{A} \ ... \in \mathcal{P}(A)}{\Gamma \vdash P <: \texttt{[}\overline{T/\texttt{X}}\texttt{]}A_i} \quad \text{(S-Import)}$$

Figure 7. Tribe$_{own}$ subtyping.

Variables may be used as types; subtyping reflects that a variable's type is a less accurate way to describe the variable than the variable itself (S-Var).

Types denote sets of objects, and a subtype is a smaller set of objects; (S-Nest-Class) and (S-Out-Nest) reflect that we can go up or down the nested class hierarchy and preserve this 'smaller set of objects' concept. (S-Out-Nest) must use substitution rather than plain nesting (that is, **[P/x]x.out** rather than P**.out**) to account for types which are semantically 'out' but do not use the **out** keyword. Furthermore, we take advantage of the definition of Tribe$_{own}$ substitution. For example, we can derive that **x.out** is a subtype of **world**.C, if we can derive that **x** is a subtype of **world**.C.D. By using (S-Out-Nest) in this way, (S-Var), and (S-Nest-Class) we can derive that any type is a subtype of its absolute type, and so the rule (S-Abs) of Tribe is admissible in Tribe$_{own}$.

Type checking Auxiliary functions are defined in Fig. 9. We define field type, and method type and body lookup functions and the lookup function $method$, which checks if a method is defined. These are all defined declaratively using the inheritance relation, as opposed to the usual inductive

Absolute types $\boxed{\Gamma \vdash P \Uparrow A}$

$$\frac{\Gamma(\gamma) = P \qquad \Gamma \vdash P \Uparrow A}{\Gamma \vdash \gamma \Uparrow A}\ (\text{A-VAR}) \qquad \frac{}{\Gamma \vdash \mathtt{world} \Uparrow \mathtt{world}}\ (\text{A-WORLD})$$

$$\frac{\Gamma \vdash p \Uparrow A.N \qquad \mathit{fType}(\mathtt{out}, A.N)\ \mathit{defined}}{\Gamma \vdash p.\mathtt{out} \Uparrow A}\ (\text{A-OUT})$$

$$\frac{\Gamma \vdash P \Uparrow A \qquad \vdash A \sqsubseteq_i^* A' \qquad \alpha\ \mathtt{class\ C\langle \overline{X}\rangle}\ \ldots \in \mathcal{P}(A') \qquad |\overline{T}| = |\overline{\mathtt{X}}| \qquad \Gamma \vdash \overline{T}\ \text{OK}}{\Gamma \vdash P.\mathtt{C}\langle\overline{T}\rangle \Uparrow A.\mathtt{C}\langle\overline{T}\rangle}\ (\text{A-CLASS})$$

Well-formed types $\boxed{\Gamma \vdash T\ \text{OK}}$

$$\frac{\Gamma \vdash P \Uparrow A}{\Gamma \vdash P\ \text{OK}}\ (\text{F-PATH-TYPE}) \qquad \frac{\mathtt{X} \in \Gamma}{\Gamma \vdash \mathtt{X}\ \text{OK}}\ (\text{F-TYPE-VAR})$$

Well-formed heaps $\boxed{\vdash \mathcal{H}\ \text{OK}}$

$$\frac{\forall \iota \to \{\iota'.\mathtt{C}\langle\overline{R}\rangle; \overline{\mathtt{f} \to v}\} \in \mathcal{H} \qquad \mathtt{f}_i = \mathtt{out} \Rightarrow v_i = \iota' \qquad \mathit{fType}(\overline{\mathtt{f}}, A) = \overline{T} \qquad \mathcal{H} \vdash \overline{v\ :\ [\iota/\mathtt{this}]T} \qquad \mathcal{H} \vdash \iota'.\mathtt{C}\langle\overline{R}\rangle \Uparrow A}{\vdash \mathcal{H}\ \text{OK}}\ (\text{F-HEAP})$$

Figure 8. Tribe$_{own}$ well-formed types and heaps.

$$\frac{\mathcal{P} = \overline{Q}\ e}{\mathcal{P}(\mathtt{world}) = \overline{Q}} \qquad \frac{\alpha\ \mathtt{class\ C\langle\overline{X}\rangle}\ \ldots\ \{\overline{Q}\ \ldots\} \in \mathcal{P}(A)}{\mathcal{P}(A.\mathtt{C}\langle\overline{T}\rangle) = [\overline{T}/\overline{\mathtt{X}}]\overline{Q}}$$

$$\frac{\vdash A \sqsubseteq_i^* A'.\mathtt{C}\langle\overline{T}\rangle \qquad \alpha\ \mathtt{class\ C\langle\overline{X}\rangle}\ \ldots\ \{\overline{Q}\ \overline{U\ \mathtt{f};}\ \overline{M}\} \in \mathcal{P}(A')}{\mathit{fType}(\mathtt{f}_k, A) = [\overline{T}/\overline{\mathtt{X}}]U_k}$$

$$\frac{\vdash A \sqsubseteq_i^* A'.\mathtt{C}\langle\overline{T}\rangle \qquad \alpha\ \mathtt{class\ C\langle\overline{X}\rangle}\ \ldots\ \{\overline{Q}\ \overline{U'\ \mathtt{f};}\ \overline{M}\} \in \mathcal{P}(A') \qquad U\ \mathtt{m}(\overline{U\ \mathtt{x}})\ \{\mathtt{return}\ e;\} \in \overline{M}}{\begin{array}{c}\mathit{mType}(\mathtt{m}, A) = [\overline{T}/\overline{\mathtt{X}}]\Pi\overline{\mathtt{x}:U}.U \\ \mathit{mBody}(\mathtt{m}, A) = (\overline{\mathtt{x}}; [\overline{T}/\overline{\mathtt{X}}]e) \\ \mathit{method}(M_k, A)\end{array}}$$

$$\frac{}{\mathit{public}(\mathtt{world})} \qquad \frac{\vdash A \sqsubseteq_i^* A' \qquad \mathtt{public\ class\ C\langle\overline{X}\rangle}\ \ldots \in \mathcal{P}(A')}{\mathit{public}(A.\mathtt{C}\langle\overline{T}\rangle)}$$

$$\frac{}{\mathcal{A}_{\mathcal{H}}(\mathtt{world}) = \mathtt{world}} \qquad \frac{\mathcal{H}(\iota) = \{r.N; \ldots\}}{\mathcal{A}_{\mathcal{H}}(\iota) = \mathcal{A}_{\mathcal{H}}(r).N}$$

Figure 9. Auxiliary functions for Tribe$_{own}$.

formulation [32] or using class tables [19, 42]. Finally, $\mathcal{A}_{\mathcal{H}}$ finds the absolute type of an address in the heap.

Rules for type checking Tribe$_{own}$ are given in Fig. 10. (T-SUBS), (T-LET), and (T-NULL) are standard. A variable (or **world**) type checks (T-VAR) if it is a valid type. A variable's minimal type is that variable itself; the type recorded in the environment can be used by subsumption. (T-FIELD), (T-ASSIGN), and (T-INVK) lookup the absolute type of the receiver and use that absolute type to look up the type of the field or method; otherwise, they are mostly standard. Of note in these rules is the substitution of the receiver for **this** (and similarly for the arguments in (T-INVK)), **this** (and arguments) may appear in Tribe$_{own}$ types and must be substituted away accordingly. (T-NEW) requires a receiver which is unusual for object-oriented languages without nested classes: it allows the precise class to be instantiated to be determined statically. We require receivers and actual parameters to method calls to be variables, this simplifies substitution. We do not lose expressivity because we can use let expressions to create local variables as needed. If we create a local variable with a path, then we can use that path as a type so the type system can be aware of the relation between variable and path.

Rules for type checking programs, classes, and methods are given in Fig. 11. The absolute type of **this** is kept up to date in the environment by (T-CLASS) and (T-PROG). Elements of a class (methods, field types, classes, superclasses, etc.) are checked to be well-formed. We also check that imported modules are in fact modules, that is, they do not have an **out** field. We check that overriding methods have matching types with all methods they override. Method type checking is standard, other than that return and argument types may include the names of formal arguments and **this**.

Operational semantics We give a large step operational semantics for Tribe$_{own}$ in Fig. 12; rules for handling error conditions are standard and are relegated to the accompanying technical report [15]. Despite the complexities of virtual class languages and ownership, all semantics rules are standard for object-oriented formalisms. In (R-NEW), we initialise fields to **null** and set the **out** field to the new object's surrounding object.

3.4 Properties

We prove preservation (subject reduction) for Tribe$_{own}$, this states that reduction preserves the type of expressions:

Expression typing $\boxed{\Gamma \vdash e : T}$

$$\frac{\Gamma \vdash e : T' \quad \Gamma \vdash T' <: T \quad \Gamma \vdash T \text{ OK}}{\Gamma \vdash e : T} \text{ (T-SUBS)}$$

$$\frac{\Gamma \vdash e : T' \quad \Gamma, \mathtt{x}{:}T' \vdash e' : T \quad \Gamma \vdash T \text{ OK}}{\Gamma \vdash \mathtt{let\ x}{:}T' = e \ \mathtt{in}\ e' : T} \text{ (T-LET)}$$

$$\frac{\Gamma \vdash \gamma \Uparrow A \quad fType(\mathtt{f}, A) = T}{\Gamma \vdash \gamma.\mathtt{f} : [\gamma/\mathbf{this}]T} \text{ (T-FIELD)}$$

$$\frac{\Gamma \vdash \gamma \Uparrow A \quad fType(\mathtt{f}, A) = T \quad \mathtt{f} \neq \mathbf{out} \quad \Gamma \vdash e : [\gamma/\mathbf{this}]T}{\Gamma \vdash \gamma.\mathtt{f} = e : [\gamma/\mathbf{this}]T} \text{ (T-ASSIGN)}$$

$$\frac{\Gamma \vdash T \text{ OK}}{\Gamma \vdash \mathtt{null} : T} \text{ (T-NULL)}$$

$$\frac{\Gamma \vdash s \text{ OK}}{\Gamma \vdash s : s} \text{ (T-VAR)}$$

$$\frac{\Gamma \vdash \gamma.N \Uparrow A.N}{\Gamma \vdash \mathbf{new}\ \gamma.N : \gamma.N} \text{ (T-NEW)}$$

$$\frac{\Gamma \vdash \gamma \Uparrow A \quad \Gamma \vdash \overline{\gamma : [\gamma/\mathbf{this}, \overline{\gamma/\mathtt{x}}]T} \quad mType(\mathtt{m}, A) = \Pi\overline{\mathtt{x}{:}T}.T}{\Gamma \vdash \gamma.\mathtt{m}(\overline{\gamma}) : [\gamma/\mathbf{this}, \overline{\gamma/\mathtt{x}}]T} \text{ (T-INVK)}$$

Figure 10. Tribe$_{own}$ expression typing rules.

Theorem: preservation $\vdash \mathcal{H}$ OK, $\mathcal{H} \vdash e : T, e; \mathcal{H} \rightsquigarrow e'; \mathcal{H}', e' \neq \mathtt{err} \Rightarrow \vdash \mathcal{H}'$ OK, $e' = v, \mathcal{H}' \vdash v : T$. Proof is by structural induction on the derivation of $e; \mathcal{H} \rightsquigarrow e'; \mathcal{H}'$.

The proof of this theorem and supporting lemmas are given in the accompanying technical report [15]; the proof structure mostly follows [19].

4. Tribal Topologies

In this section we will show how Tribe$_{own}$ can be used as a descriptive [14, 16] ownership system 'for free' — that is, without any additional program annotations or formal rules. In the next section we will show how a range of different prescriptive ownership policies can be supported by the same underlying descriptive Tribe$_{own}$ type system — separating ownership policies from the mechanisms that implement them [1].

The key insight is that the nesting of classes in objects, used in Tribe$_{own}$ to support family polymorphism, is precisely the information needed to support descriptive ownership. Lexically, classes are nested in other classes; semantically, classes, and their instantiations, are nested under an

Program typing $\boxed{\vdash \mathcal{P} \text{ OK}}$

$$\frac{\mathbf{this}{:}\mathbf{world} \vdash \overline{Q \text{ OK}} \quad \mathbf{this}{:}\mathbf{world} \vdash e : T \quad \sqsubseteq_i^+ \ is\ acyclic}{\vdash \overline{Q}\ e \text{ OK}} \text{ (T-PROG)}$$

Class typing $\boxed{\Gamma \vdash Q \text{ OK}}$

$$\frac{\begin{array}{c}\Gamma(\mathbf{this}) = A' \quad \Gamma' = \Gamma[\mathbf{this} \mapsto A'.\mathtt{C}\langle\overline{\mathtt{X}}\rangle], \overline{\mathtt{X}} \\ \Gamma' \vdash A'.\overline{N}, \overline{A}, \overline{Q}, \overline{T}, \overline{M} \text{ OK} \\ \forall A \in \overline{A}.\ fType(\mathbf{out}, A)\ \textit{undefined} \\ \mathtt{f}_i = \mathbf{out} \Rightarrow T_i = \mathbf{this.out} \\ \forall M, M' \in \{M | method(M, A'.\mathtt{C}\langle\overline{\mathtt{X}}\rangle)\}.\ override(M, M') \\ \alpha = \mathtt{private} \Rightarrow \forall A\ st \vdash A'.\mathtt{C}\langle\overline{\mathtt{X}}\rangle \sqsubseteq_i^*\ A.\ \neg public(A)\end{array}}{\Gamma \vdash \alpha\ \mathtt{class\ C}\langle\overline{\mathtt{X}}\rangle \lhd \overline{N} \bigtriangledown \overline{A}\ \{\overline{Q}\ \overline{T\,\mathtt{f};}\ \overline{M}\} \text{ OK}} \text{ (T-CLASS)}$$

Method typing $\boxed{\Gamma \vdash M \text{ OK}}$

$$\frac{\Gamma, \overline{\mathtt{x}{:}T} \vdash e : T \quad \Gamma, \overline{\mathtt{x}{:}T} \vdash \overline{T}, T \text{ OK}}{\Gamma \vdash T\,\mathtt{m}(\overline{T\,\mathtt{x}})\ \{\mathtt{return}\ e;\} \text{ OK}} \text{ (T-METHOD)}$$

$$\frac{M = T\,\mathtt{m}(\overline{T\,\mathtt{x}})\ \{\mathtt{return}\ e;\} \quad M' = T\,\mathtt{m}(\overline{T\,\mathtt{x}})\ \{\mathtt{return}\ e';\}}{override(M, M')} \text{ (T-OVERRIDE)}$$

$$\frac{M = T\,\mathtt{m}(\overline{T\,\mathtt{x}})\ \{\mathtt{return}\ e;\} \quad M' = T'\,\mathtt{m}'(\overline{T'\,\mathtt{x}'})\ \{\mathtt{return}\ e';\} \quad \mathtt{m} \neq \mathtt{m}'}{override(M, M')} \text{ (T-OVERRIDE-NE)}$$

Figure 11. Tribe$_{own}$ typing rules for classes and methods.

object: the owner of those instantiations. This nesting of objects is our ownership hierarchy.

The owner of an object (or an approximation to the owner) can be derived from its type. For object family types, the owner is the path part, for example, the owner of `x.C` is `x`, the owner of `this.out.C` is `this.out`, and the owner of `world.C` is `world`. Variables with singleton (path) types can only hold the object indicated by the singleton type, therefore the owner of objects with singleton type is the owner of the path; e.g., the owner of objects with type `x.out` is `x.out.out`.

If the type is a variable, then we can find a more precise owner by examining the type of the variable. For example, if `x` has type `y.C`, then an object with type `x` will have owner `y` or `x.out`, and these two paths will always denote the same object. We know they denote the same object because they are subtypes (`y` is a subtype of `x.out` by rules (S-

Computation $\boxed{e; \mathcal{H} \rightsquigarrow v; \mathcal{H}}$

$$\frac{}{v; \mathcal{H} \rightsquigarrow v; \mathcal{H}} \text{(R-VAL)} \qquad \frac{\mathcal{H}(\iota) = \{T; \overline{\mathtt{f} \rightarrow v}\}}{\iota.\mathtt{f}_k; \mathcal{H} \rightsquigarrow v_k; \mathcal{H}} \text{(R-FIELD)}$$

$$\frac{e; \mathcal{H} \rightsquigarrow v; \mathcal{H}' \qquad [v/\mathtt{x}]e'; \mathcal{H}' \rightsquigarrow v'; \mathcal{H}''}{\mathtt{let\ x}{:}T = e\ \mathtt{in}\ e'; \mathcal{H} \rightsquigarrow v'; \mathcal{H}''} \text{(R-LET)}$$

$$\frac{\overline{\mathtt{f}} = \{\mathtt{f} | fType(\mathtt{f}, \mathcal{A}_{\mathcal{H}_n}(r).N)\ \textit{defined}\} \qquad \mathcal{H}' = \mathcal{H}_n, \iota \rightarrow \{r.N; \mathtt{out} \rightarrow r, \overline{\mathtt{f} \rightarrow \mathtt{null}}\}}{\mathtt{new}\ r.N; \mathcal{H} \rightsquigarrow \iota; \mathcal{H}'} \text{(R-NEW)}$$

$$\frac{e; \mathcal{H} \rightsquigarrow v; \mathcal{H}_1 \qquad \mathcal{H}_1(\iota) = \{R; \overline{\mathtt{f} \rightarrow v}\} \qquad \mathcal{H}' = \mathcal{H}_1[\iota \mapsto \{R; \overline{\mathtt{f} \rightarrow v}[\mathtt{f}_k \mapsto v]\}]}{\iota.\mathtt{f}_k = e; \mathcal{H} \rightsquigarrow v; \mathcal{H}'} \text{(R-ASSIGN)}$$

$$\frac{\overline{e; \mathcal{H} \rightsquigarrow r; \mathcal{H}_n} \qquad mBody(\mathtt{m}, \mathcal{A}_{\mathcal{H}_n}(\iota)) = (\overline{\mathtt{x}}; e) \qquad [\iota/\mathtt{this}, \overline{r/\mathtt{x}}]e; \mathcal{H}_n \rightsquigarrow v; \mathcal{H}'}{\iota.\mathtt{m}(\overline{e}); \mathcal{H} \rightsquigarrow v; \mathcal{H}'} \text{(R-INVOKE)}$$

Figure 12. Tribe$_{own}$ reduction rules.

Owner relation $\boxed{\Gamma \vdash P \Downarrow P}$

$$\frac{\Gamma \vdash P \Uparrow A \qquad fType(\mathtt{out}, A)\ \textit{defined}}{\Gamma \vdash P \Downarrow [P/\mathtt{x}]\mathtt{x.out}} \text{(O-OWNER)}$$

Inside relation $\boxed{\Gamma \vdash P \prec: P}$

$$\frac{\Gamma \vdash P <: P' \qquad \Gamma \vdash P' \Uparrow A \qquad fType(\mathtt{out}, A)\ \textit{defined}}{\Gamma \vdash P \prec: P'} \text{(I-SUB)} \qquad \frac{\Gamma \vdash P \prec: P'' \qquad \Gamma \vdash P'' \prec: P'}{\Gamma \vdash P \prec: P'} \text{(I-TRANS)}$$

$$\frac{}{\Gamma \vdash p \prec: p.\mathtt{out}} \text{(I-OUT)} \qquad \frac{}{\Gamma \vdash P.N \prec: P} \text{(I-CLASS)}$$

$$\frac{\mathcal{H}(\iota) = \{r.N; ...\}}{\mathcal{H} \vdash \iota \prec: r} \text{(I-RUNTIME)}$$

Figure 13. Tribe$_{own}$ owner function and inside relation.

VAR) and (S-OUT-NEST)) and *path types* that are subtypes always denote the same object (which is obvious since their interpretation are singleton sets, and the interpretation of subtyping is subset).

For other types, we cannot say what the precise owner is. This is clear if we consider, for example, objects with type `x.Course.RawMark` that denote any `RawMark` object in any `Course` object in `x`. We say that the *owner type* of `x.Course.RawMark` is `x.Course`.

In general, an owner type in Tribe$_{own}$ can be any non-variable type; if the owner type is not a path, then it does not denote a specific owner. Non-path owner types represent partial information about the actual owner, and are similar to existential [14, 49] or variant owners [35]. Subtyping between owner types can be used to identify types that refer to the same owner or set of owners, or refer to a more precise owner type. For example, if `x` is a subtype of `y.C` then the owner type `x` is a single owner and `y.C` describes a group of owners which includes `x`.

An owner type is not defined for all types. Because module classes can be imported into any other class, objects with module type may have any owner. Therefore, types which indicate a module class do not have defined owner types; for example, **`world`**`.Collections` cannot be assumed to have owner **`world`**, because it may be imported into some other class with a different owner; if **`world`**`.Collections` is imported into some non-module class `C` with owner `x` then `x.Collections` will have owner `x`, as usual. Note also that **`world`** does not have an owner, because it is the top level of nesting.

We use the judgement $\Gamma \vdash T \Downarrow P$ to find the owner (P) from a type (T). The owner judgement is defined in Fig. 13. Within a class, **`out`** can be used to name the current object's owner in types and expressions. We find the owner by leveraging Tribe$_{own}$ substitution: we attempt to append **`out`** and Tribe$_{own}$ substitution gives us the result described informally above. We must, of course, check that **`out`** is defined in the given type.

At runtime, we can find an object's owner by observing its object record in the heap. The object record stores an object's runtime type which is an object family type, the owner can be found in the usual way. The owner can also be found by looking at the value in the object's **`out`** field, these two approaches will give the same result in a well-formed heap.

By observing that each object in the heap is defined as having a single owner, and that the resulting owner relation between objects in the heap is acyclic (because the owner must be present in the heap at the time an object with that owner is created), we deduce that owners in the heap form a tree. Therefore, our ownership system describes an hierarchical topology for the heap. We prove below that the topology described statically by the type system is sound with respect to the runtime heap topology.

4.1 The Inside Relation

We define the inside relation [40] for owner types in Tribe$_{own}$ in Fig. 13. The inside relation is reflexive (given by (I-SUB) and (S-REFLEX)) and transitive. Since subtyping indicates more precise descriptions of the same owner-type, subtyping between two owner types gives an inside relation[4]. Rules (I-OUT) and (I-CLASS) describe stepping outward or inward one step in the nested class hierarchy, which corresponds to one step outward or inward in the ownership hierarchy. (I-RUNTIME) is used to compute the inside relation for dynamic owner types and is based directly on the structure of the heap.

4.2 Imported Module Classes

Having owners defined by the lexical nesting of classes could limit reuse, for example, a programmer would need to write a new list class nested inside every class which needs to own a list. Reuse of classes by importing modules is thus crucial for creating a practical ownership system. Importing a module not only allows the code to be reused, but also allows the importing class to own the imported classes. To the best of our knowledge, this feature is not found in other virtual classes languages. For example,

```
module Collections {
   class Map<K, V> {...}
}

class University {
  ...
  class Course imports Collections{
    ...
    this.Map<this.out.Student,
      this.RawMark> marks_map = ...;
  }
}
```

Here, the field `marks_map` is an instance of `Map` owned by the current instance of `Course`.

4.3 Discussion

In most ownership systems, an object may be owned by any other object in the program and the owner is independent of the object's class. Classes are declared in a flat topology and there is no direct indication of the ownership hierarchy in the source code. In Tribe$_{own}$, an object's place in the ownership hierarchy is determined in part by the hierarchy of nested classes. Although the owner of each object may be different, the class of the owner is determined by the owned object's class. That is, while objects of the same class may be owned by different owners, the objects will be on the same tier of the ownership hierarchy and their owners will have a single class. Furthermore, objects of different classes are guaranteed to have different owners. Analogously to variance annotations [13], more of the ownership information in Tribe$_{own}$ is defined where a class is declared, compared to other ownership systems where all ownership information is defined where classes are used.

In Tribe$_{own}$ the nesting of classes specifies the topology of objects in the same way as individual classes specify the structure and behaviour of individual objects in class-based object-oriented languages.

4.4 Tribal Effects

To show that our descriptive ownership system is useful, we have used it to define an effect system [45] for Tribe$_{own}$. The nested-ness of Tribe$_{own}$ is used to define computational effects without extra ownership or regions notations. This should make effects systems accessible and attractive to programmers. We offer the formalised effect system without proofs in our technical report [15].

Our effects can describe a single object (by using a singleton type and an 'exact' annotation), a single object and its transitive representation (by using a singleton type), a context (exact, non-singleton type), or a context and all contexts under it (non-singleton types). Effects can be combined. The sub-effect relation follows from the subtype and inside relations. Our effect assignment rules are mostly standard and we expect the usual rules for disjointness of effects [18] to apply. Therefore, Tribal effects could be used to support parallelisation of Tribe$_{own}$ code [18, 5].

4.5 Properties

We prove that ownership in Tribe$_{own}$ is sound; that is, the owner given by an expression's static type corresponds to the owner, found in the heap, after evaluating that expression. This result is necessary for supporting an effect system or other application of descriptive ownership.

Theorem: Reduction preserves owners $\vdash \mathcal{H}$ OK, $\mathcal{H} \vdash e : T, e; \mathcal{H} \rightsquigarrow \iota; \mathcal{H}', \mathcal{H} \vdash T \Downarrow P \Rightarrow \mathcal{H}'(\iota) = \{r.\texttt{C<...>}; ...\}, \mathcal{H}' \vdash r <: P$

This theorem follows straightforwardly from our preservation theorem (see Sect. 3.4) and the following lemma.

Lemma: Subtyping preserves owners $\Gamma \vdash P <: P', \Gamma \vdash P' \Downarrow P'_0 \Rightarrow \Gamma \vdash P \Downarrow P_0, \Gamma \vdash P_0 <: P'_0$. Proof is by structural induction on the derivation of the subtyping judgement.

In Tribe$_{own}$ different paths can point to the same object, thus a single owner can be denoted by different types. We formalise this fact using subtyping: p and p' denote the same object if and only if p is a subtype of p' or p' is a subtype of p. This statement only makes sense if subtyping monotonically increases precision of types, we prove this as a lemma to support our ownership soundness argument:

[4] In fact, it indicates a 'reflexive' relationship with respect to the inside relation, if we offered a 'single step' (irreflexive, intransitive) version of the inside relation, then the rule (I-SUB) could not be used.

Lemma: Subtypes are more precise $\Gamma \vdash T <: p \Rightarrow T = p' \; and \; \Gamma \vdash T <: \gamma.N \Rightarrow T \in \{p, p.N'\}$. Proof is by structural induction on the derivation of the subtyping judgement.

5. Encapsulation

In this section, we show how encapsulation policies can be enforced in Tribe$_{own}$ by building on top of the descriptive ownership outlined in the previous section. We construct three different variants of owners-as-modifiers (in Fig. 14) and a new, more flexible variant of owners-as-dominators with two mechanisms for enforcement (in Fig. 15); we show how standard owners-as-dominators is a simple restriction of our variant. Encapsulation policies in Tribe$_{own}$ are defined independently of the mechanism of descriptive ownership, and are, essentially, pluggable.

Owners-as-Modifiers (syntactic)

$$\frac{\Gamma \vdash \gamma : \texttt{this}.N^* \quad \ldots}{\Gamma \vdash \gamma.\texttt{f} = e : \ldots \quad \Gamma \vdash \gamma.\texttt{m}(\overline{e}) : \ldots}$$

(T-ASSIGN T-INVK)

Owners-as-Modifiers (expressive)

$$\frac{\Gamma \vdash \gamma \Downarrow P \qquad \Gamma \vdash P \prec: \texttt{this} \vee P = \texttt{this.out} \quad \ldots}{\Gamma \vdash \gamma.\texttt{f} = e : \ldots \quad \Gamma \vdash \gamma.\texttt{m}(\overline{e}) : \ldots}$$

(T-ASSIGN T-INVK)

Owners-as-Modifiers (Universes)

$$\frac{\Gamma \vdash \gamma \Downarrow P \qquad P \in \{\texttt{this}, \texttt{this.out}\} \quad \ldots}{\Gamma \vdash \gamma.\texttt{f} = e : \ldots \quad \Gamma \vdash \gamma.\texttt{m}(\overline{e}) : \ldots}$$

(T-ASSIGN T-INVK)

Figure 14. Enforcing owners-as-modifiers in Tribe$_{own}$.

5.1 Owners-as-Modifiers

Owners-as-modifiers is an encapsulation policy first implemented in the Universes type system [39, 27, 23]. It enforces that any modification of an object must be initiated by its owner.

Owners-as-modifiers in the Tribal setting is implemented by adding restrictions (extra premises) to the rules for field assignment and method invocation. For simplicity, we ignore pure methods. They could easily be supported by allowing all calls to pure methods; the formalisation is standard (e.g., [23]).

Owners-as-Local-Dominators

$$\frac{\Gamma \vdash \texttt{this} \oplus \overline{T} \quad \ldots}{\Gamma \vdash \alpha \; \texttt{class C<}\overline{\texttt{X}}\texttt{>} \lhd \overline{A} \bigtriangledown \overline{\texttt{C}'} \; \{\overline{Q} \; \overline{T \; \texttt{f;}} \; \overline{M}\} \; \text{OK}}$$

(T-CLASS)

$$\frac{\ldots \quad \Gamma \vdash P.\texttt{C<}\overline{T}\texttt{>} \oplus \overline{T}}{\Gamma \vdash P.\texttt{C<}\overline{T}\texttt{>} \Uparrow A.\texttt{C<}\overline{T}\texttt{>}}$$

(A-CLASS)

Owners-as-Local-Dominators Heap

$$\frac{\forall \iota \rightarrow \{R; \overline{\texttt{f} \rightarrow v}\} \in \mathcal{H} \quad \ldots \quad \forall \texttt{f} \rightarrow v \in \overline{v}.\; \texttt{f} \neq \texttt{out} \wedge v \neq \texttt{null} \Rightarrow \Gamma \vdash \iota \prec: v.\texttt{out} \vee (\mathcal{H} \vdash \mathcal{H}(v) \Uparrow A \wedge public(A))}{\vdash \mathcal{H} \; \text{OK}}$$

(F-HEAP)

Figure 15. Enforcing owners-as-dominators in Tribe$_{own}$.

Over the next few paragraphs we describe three variations of the owners-as-modifiers discipline. Each variation is implemented in a different way and enforces a slightly different invariant; all three variations are defined formally in Fig. 14.

Syntactic variant Our "syntactic variant" of owners-as-modifiers has a simple implementation, formalised by examining the syntax of the static type of the receiver. We only allow field assignment and method invocations on objects nested inside the current **this**.

The syntactic variant allows an aggregate to manipulate its (transitive) representation (instances of classes nested, possibly deeply, inside the aggregate), but not operate on peer objects (instances of classes directly nested inside the aggregate's owner).

Expressive variant The "expressive variant" is, to the best of our efforts, the most flexible policy that sticks to the spirit of owners-as-modifiers. It allows access to transitively owned objects and peer objects. It is formalised using the owner and inside relations, rather than a syntactic test, and is therefore more expensive to implement. It handles paths which alias a transitively owned object. Finally, it subsumes the simple variant.

Universes variant The "Universes variant" emulates Universe Types. An object can modify its own representation and peer objects, but nothing else. The formalisation uses the

Naming $\boxed{\Gamma \vdash P \oplus T}$

Local o-as-d, semantic style

$$\frac{\Gamma \vdash P' \Downarrow P'' \qquad \Gamma \vdash P \prec: P''}{\Gamma \vdash P \oplus P'} \text{(N-O-AS-D)} \qquad \frac{}{\Gamma \vdash P \oplus \mathtt{X}} \text{(N-TYPE-VAR)}$$

$$\frac{\Gamma \vdash T \Uparrow A \qquad public(A.N) \qquad \Gamma \vdash P \oplus T}{\Gamma \vdash P \oplus T.N} \text{(N-PUBLIC)}$$

Local o-as-d, syntactic style

$$\frac{\Gamma \vdash T \Uparrow A \qquad public(A.N) \qquad \Gamma \vdash P \oplus T}{\Gamma \vdash P \oplus T.N} \text{(N-PUBLIC)} \qquad \frac{}{\Gamma \vdash P \oplus \mathtt{X}} \text{(N-TYPE-VAR)}$$

$$\frac{}{\Gamma \vdash p.N \oplus p.N} \quad \frac{}{\Gamma \vdash p \oplus p} \text{(N-REFLEX)} \qquad \frac{}{\Gamma \vdash p \oplus p.N} \text{(N-REP)}$$

$$\frac{\Gamma \vdash P \oplus p}{\Gamma \vdash P \oplus p.\mathtt{out}} \quad \frac{}{\Gamma \vdash P \oplus p.\mathtt{out}.N} \text{(N-OUT-1)} \qquad \frac{}{\Gamma \vdash p.N \oplus p} \text{(N-OUT-2)}$$

$$\frac{\Gamma \vdash P' \oplus T \qquad \Gamma \vdash P <: P'}{\Gamma \vdash P \oplus T} \text{(N-SUB)}$$

Figure 16. Tribe$_{own}$ naming restrictions.

owner relation, but not the inside relation (and thus, not subtyping), it is therefore 'more syntactic' than the expressive variant, but not purely so. Similarly to the syntactic variant, the Universes variant only handles types which start at **this**, not equivalent types.

5.2 Owners-as-Local-Dominators

In a system that supports owners-as-dominators, an object's owner is found on all access paths from the root of the system to the object itself [22]. In this section we propose a new, more flexible variation of owners-as-dominators which has two formalisations. We call our policy owners-as-local-dominators; it subsumes owners-as-dominators.

The rules in figures 15 and 16 define owners-as-local-dominators encapsulation in Tribe$_{own}$. By (T-CLASS), a class may only use types that it is allowed to name. Naming, denoted $\oplus$, is defined in two ways: a *semantic variant* with few, declarative rules and a *syntactic variant* with more rules, but using only the syntax of types, where possible.

We call our policy owners-as-local-dominators because it enforces owners-as-dominators over local sub-heaps, rather than the whole heap. This is supported in the language by declaring classes **public** or **private**. Public class names can be named anywhere in a program, private class names can only be named within the surrounding class's declaration (at any level of nesting). These private naming rules mimic standard ownership types, where naming is dictated by the visibility of **this**. As a consequence, each instance of a private class forms the root of an encapsulated sub-heap. Objects in the sub-heap may be referenced only according to the owners-as-dominators policy.

In both our naming formalisations, the rules for public classes and type variables are the same. Both sets of rules handle aliasing paths by subtyping, in the semantic version via (I-SUB), and explicitly in the syntactic version.

A public class nested inside an encapsulated sub-heap may be referenced freely *within that heap*, but not outside. It is easy to see why this is the case: WLOG, assume p.C is a type where p is a path to an object with private class and C is a public class name. By the rules in Fig. 16, to name p.C, an object must be able to name p. As a result, public classes provide flexibility without compromising safety for objects inside an aggregate. Importantly, it is not possible to define a public class inside an encapsulated sub-heap and use it as a proxy to export references to objects outside the sub-heap[5].

For concreteness, consider the following code:

```
private class A {
    private class B {
        private class C {}
        public class D {}
    }
    private class E {
        this.out f1; //OK - owner
        this.out.B f2; //OK - peer
        this.out.B.C f3; //ERROR - private class
        this.out.B.D f4; //OK - can name B
        world.F f5; //OK - public class
    }
}
public class F {
    world.A f6; //OK - peer
    world.A.B.D //ERROR can't name B
}
```

Public classes may inherit from private classes, but not vice versa. Therefore, instances of private classes cannot be named by subsumption. It is safe to subsume a public class to a private class because a public class can be named everywhere that a private class can.

Because we wish to use Root.Object as a top type, we must be able to name it everywhere, and so it must be a public class. Since private classes can not inherit from public classes, we require a Root.PrivateObject to be the root of

[5] This distinguishes deep ownership from shallow ownership.

the inheritance hierarchy for private classes[6]. Unfortunately, this means `Root.Object` is not a top type after all, but only one for public classes. We believe this is a feature rather than a flaw, as subsuming private classes to a global top type effectively breaks encapsulation.

If all classes are private (or we remove the rule (N-PUBLIC)), our system upholds the owners-as-dominator property. If all classes are public, then we enforce no encapsulation policy and have descriptive ownership. By using a mix of public and private classes, we allow the programmer to define the level of encapsulation required.

For increased flexibility we only enforce our encapsulation policy *on the heap*. Similar to Clarke and Drossopoulou's Joe$_1$ [18], we do not enforce naming restrictions on the types of local variables. (See also a discussion of borrowing and owner-polymorphic methods in [20].)

Owners-as-local dominators and a default **`public`** annotation for classes makes it easy to allow arbitrary aliasing in a system; except for specific sub-heaps which can be easily created by a single **`private`** annotation on a class. This is similar to the flat ownership hierarchies in Joelle [21] and Loci [50], where fields are implicitly encapsulated in the current owning object, and **`active`** and `@Thread` annotations are used as "cut-off points" to define the root of an encapsulated sub-heap. The big difference is that in Tribe$_{own}$, the ownership hierarchy is not flat, even when aliasing is permitted, which facilitates gradually increasing encapsulation.

5.2.1 Properties

We define $oasld(\mathcal{H})$ if $\forall \iota, \iota' \in \mathcal{H}\ where\ \mathcal{H} \vdash \iota \hookrightarrow \iota'.\ \Gamma \vdash \iota \prec: \iota'.\mathbf{out} \vee (\mathcal{H} \vdash \mathcal{H}(\iota') \Uparrow A \wedge public(A))$, where $\hookrightarrow$ means has a reference to.

Theorem: owners-as-local-dominators $\vdash \mathcal{H}$ OK, $\mathcal{H} \vdash e : T, e; \mathcal{H} \rightsquigarrow e'; \mathcal{H}', oasld(\mathcal{H}) \Rightarrow oasld(\mathcal{H}')$

We prove this theorem for both syntactic and semantic variants of our naming relation. Both are proved as part of our preservation theorem by proving that the extra invariants in (F-HEAP) are preserved. This requires extending our substitution lemmas to the inside, owner, and naming relations, and, in the syntactic case, proving that syntactic naming gives semantic naming.

5.2.2 Discussion

Superficially, owners-as-local-dominators appears to be orthogonal to the family-polymorphic aspects of Tribal ownership. The programmer can specify encapsulated sub-heaps by annotating classes, however, only because of the unique correspondence between the nested class hierarchy and the ownership hierarchy. In a context-parametric ownership system, it would only make sense to annotate individual context parameters. This is not a solution however, because we could not control the nesting of public and private contexts; therefore, this would not create encapsulated sub-heaps.

[6] `PrivateObject` can be inherited from because its surrounding class is public. `Root.PrivateObject` cannot be used as a supertype of the local `PrivateObject`s because it cannot be named.

6. Related Work

Tribe$_{own}$ is closest in spirit to work by Dietl and Müller [26] expressing ownership type systems on top of dependent classes (a similar language feature to virtual classes, but that allows classes to be nested inside more than one enclosing object). Their system is inspired by the original Tribe work and has similar properties to Tribe$_{own}$—topological constraints and encapsulation restrictions are handled separately. Their system is an encoding and has not been formalised or proven correct. Furthermore, it requires more annotations (notably, special owner annotations) than our system. Thus, we believe that Tribe$_{own}$ is a further simplification of Dietl and Müller's system. Tribe$_{own}$ also comes with proofs of type and ownership soundness.

The simplest way to support ownership in a Tribe-like language is to mimic the context parameters of context-parametric ownership systems with fields. For example, a non-generic map could be written as:

```
class Map {
  Object keyOwner;
  Object valueOwner;
  this.Node[] nodes;

  Map(Object keyOwner, Object valueOwner) {...}


  class Node {
    this.out.keyOwner.Object key;
    this.out.valueOwner.Object value;
  }
}

Map m = new Map(k, v);
k.Object ko = new k.Object();
m.put(ko, new v.Object()); //type error
v.Object vo = m.get(ko); //type error
```

Although, this code looks sensible, it cannot be type checked because the type system cannot determine that `m`'s `keyOwner` is the same object as `k`, and similarly for `valueOwner`. This makes the map essentially useless if we want to preserve ownership information. We address this problem in Tribe$_{own}$ by using generics. In a fully dependent class system such as Dietl and Müller's, `Map` can be dependent upon `keyOwner` and `valueOwner`, as well as the map's owner, allowing programmers to write this kind of code.

Tribe$_{own}$ uses similar restrictions to Confined Types [31] — a simple set of rules to guarantee that instances of package scoped classes defined in a package are not referenced outside the package. In confined type, there is no nesting or notion of topology. In this respect, the encapsulation pro-

vided by confined types is similar to systems that offer flat ownership hierarchies, like Joelle [21] and Loci [50].

Ownership and Alias Management Systems There are a plethora of ownership systems to date, notably Clarke's original deep ownership system [22], Universes [39], shallow ownership [3], and Ownership Domains [1]. Ownership Domains was the first system to advocate separating policy from topology. Ownership Domains is more flexible than Tribe, but requires more annotations. Since the Ownership Domains programmer can customise encapsulation, the type system will only enforce ad hoc encapsulation, not an encapsulation policy. Therefore, compilers and programmers cannot rely on the invariants of a policy to aid reasoning. In similar spirit, Boyland et al. [11] define a set of capabilities on pointers, although without a static type system.

Tribe$_{own}$'s `out` field is similar to Pedigree Types' chains of `parents` for transitive owners, although Pedigree Types are not based on virtual or nested classes [34].

Inner classes in Java are a primitive kind of nested class, lacking virtual inheritance. Boyapati et al. [8] have proposed integrating inner classes with a typical context-parametric ownership types system. In their system, the ownership hierarchy is defined orthogonally to the nested class hierarchy: inner classes have owners independent of their surrounding classes. Their system supports a modified owner-as-dominators encapsulation policy where, as well as the usual rules, inner classes have privileged access to the representations of their enclosing objects. Our owners-as-dominators policy is similar to the inner classes part of Boyapati et al.'s policy. Our policy is more flexible because nested classes have access to their peer and owned classes, as well as their owners, and more intuitive because of the coincidence of the ownership and nested class hierarchies. Our scheme of public classes introduces further flexibility so we avoid the need for having a separate ownership hierarchy.

Virtual Class Systems Since Ernst's original proposal [29], built on gBeta [28], a range of different proposals for virtual class systems have appeared, both with *class families* (Concord [33], .FJ [44], and Jx [41]) and *object families* (Caesar/J [38], *vc* [30], Scala [36], gBeta [28], and Tribe). For ownership types, object families are needed since encapsulation is per-object. Building an ownership system on top of Scala is an interesting direction for future work.

Recently, Bach Nielsen and Ernst have investigated VM support for virtual classes [4]. We believe that having efficient VM support for virtual classes could enable space-efficient downcasting in ownership types, which is a long-standing problem in the community (see further [6, 49]).

Bracha's Newspeak [12] is a recent, dynamically typed language with virtual classes. Like the aforementioned languages, Newspeak does not have ownership support.

7. Conclusion and Further Work

We have shown how Tribe$_{own}$, a language with virtual classes, can support ownership types, and a range of different encapsulation policies, in a flexible and straightforward way. We have shown that using a class's enclosing object as the owner of instances of that class gives a sound ownership system. This has been suggested before (first, to our knowledge, by Clarke [17], later by Dietl and Müller [26]) but not developed, formalised, or proven. A language based on Tribe$_{own}$ will have low syntactic overhead, and should make it easy for programmers to comprehend the ownership hierarchy. We have further described a novel, flexible encapsulation policy, owners-as-local-dominators; an extension to the standard owners-as-dominators policy, which leverages the nested-class-based ownership of Tribe$_{own}$ and allows the programmer to customise the level of encapsulation enforced.

Future work We have shown how both an owners-as-modifiers policy and an owners-as-dominator policy can be layered on top of Tribe$_{own}$, and how the latter can be localised. We did not find an obvious way to localise owners-as-modifiers. We hope to find a way to do this, then combine the local variants of both properties in the same program.

We believe that class invariants can be extended to families of classes to describe invariants which depend on the relationships between classes as well as the classes themselves; for example, to verify subject-observer [43].

Acknowledgments

We would like to thank Dave Clarke and Sophia Drossopoulou, who were part of the team defining the original Tribe system, and were also present in several initial discussions about using Tribe to express ownership types. We would also like to thank the OOPSLA reviewers for their useful feedback. The first author's work was funded in part by a Build IT postdoctoral fellowship.

References

[1] J. Aldrich and C. Chambers. Ownership Domains: Separating Aliasing Policy from Mechanism. In *European Conference on Object Oriented Programming (ECOOP)*, 2004.

[2] J. Aldrich, C. Chambers, and D. Notkin. ArchJava: Connecting Software Architecture to Implementation. In *International Conference on Software Engineering (ICSE)*, 2002.

[3] J. Aldrich, V. Kostadinov, and C. Chambers. Alias Annotations for Program Understanding. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2002.

[4] A. Bach Nielsen and E. Ernst. Virtual Class Support at the Virtual Vachine Level. In *Workshop on Virtual Machines and Intermediate Languages (VMIL)*, 2009.

[5] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A Type and Effect System for Deterministic Parallel Java. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2009.

[6] C. Boyapati, R. Lee, and M. Rinard. Safe Runtime Downcasts With Ownership Types. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, 2001.

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

[8] C. Boyapati, B. Liskov, and L. Shrira. Ownership Types for Object Encapsulation. In *Principles of Programming Languages (POPL)*, 2003.

[9] C. Boyapati, B. Liskov, L. Shrira, C.-H. Moh, and S. Richman. Lazy Modular Upgrades in Persistent Object Stores. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2003.

[10] C. Boyapati, A. Salcianu, W. Beebee, Jr., and M. Rinard. Ownership Types for Safe Region-based Memory Management in Real-time Java. In *Programming Language Design and Implementation (PLDI)*, 2003.

[11] J. Boyland, J. Noble, and W. Retert. Capabilities for Sharing: A Generalization of Uniqueness and Read-Only. In *European Conference on Object Oriented Programming (ECOOP)*, 2001.

[12] G. Bracha, P. Ahé, V. Bykov, Y. Kashai, W. Maddox, and E. Miranda. Modules as Objects in Newspeak. In *European Conference on Object Oriented Programming (ECOOP)*, 2010.

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

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

[15] N. Cameron, T. Wrigstad, and J. Noble. Tribal Ownership (accompanying technical report). Technical Report 10–14, School of Engineering and Computer Science, Victoria University of Wellington. `https://ecs.victoria.ac.nz/twiki/pub/Main/TechnicalReportSeries/ECSTR10-14.pdf`.

[16] N. R. Cameron, S. Drossopoulou, J. Noble, and M. J. Smith. Multiple Ownership. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2007.

[17] D. Clarke. Nested Classes, Nested Objects and Ownership. Invited talk at FOOL/WOOD, 2006.

[18] D. Clarke and S. Drossopoulou. Ownership, Encapsulation, and the Disjointness of Type and Effect. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2002.

[19] D. Clarke, S. Drossopoulou, J. Noble, and T. Wrigstad. Tribe: A Simple Virtual Class Calculus. In *Aspect-Oriented Software Development (AOSD)*, 2007.

[20] D. Clarke and T. Wrigstad. External Uniqueness is Unique Enough. In *European Conference on Object Oriented Programming (ECOOP)*, 2003.

[21] D. Clarke, T. Wrigstad, J. Östlund, and E. B. Johnsen. Minimal Ownership for Active Objects. In *Asian Symposium on Programming Languages and Systems (APLAS)*, 2008.

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

[23] D. Cunningham, W. Dietl, S. Drossopoulou, A. Francalanza, P. Müller, and A. J. Summers. Universe Types for Topology and Encapsulation. In *Formal Methods for Components and Objects (FMCO)*, 2008.

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

[25] W. Dietl and P. Müller. Universes: Lightweight Ownership for JML. *Journal of Object Technology*, 4(8), 2005.

[26] W. Dietl and P. Müller. Ownership Type Systems and Dependent Classes. In *Foundations of Object-Oriented Languages (FOOL)*, 2008.

[27] W. M. Dietl. *Universe Types: Topology, Encapsulation, Genericity, and Tools*. PhD thesis, ETH Zurich, Switzerland, 2009.

[28] E. Ernst. Propagating Class and Method Combination. In *European Conference on Object Oriented Programming (ECOOP)*, 1999.

[29] E. Ernst. Family Polymorphism. In *European Conference on Object Oriented Programming (ECOOP)*, 2001.

[30] E. Ernst, K. Ostermann, and W. R. Cook. A Virtual Class Calculus. In *Principles of Programming Languages (POPL)*, 2006.

[31] C. Grothoff, J. Palsberg, and J. Vitek. Encapsulating Objects with Confined Types. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(6), 2007.

[32] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a Minimal Core Calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(3), 2001.

[33] P. Jolly, S. Drossopoulou, C. Anderson, and K. Ostermann. Simple Dependent Types: Concord. In *ECOOP Workshop on Formal Techniques for Java-like Languages (FTfJP)*, 2004.

[34] Y. D. Liu and S. Smith. Pedigree Types. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, 2008.

[35] Y. Lu and J. Potter. On Ownership and Accessibility. In *European Conference on Object Oriented Programming (ECOOP)*, 2006.

[36] M. Odersky et al. An Overview of the Scala Programming Language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[37] O. L. Madsen and B. Moller-Pedersen. Virtual Classes: A Powerful Mechanism in Object-oriented Programming. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1989.

[38] M. Mezini and K. Ostermann. Conquering Aspects with Caesar. In *Aspect-Oriented Software Development (AOSD)*, 2003.

[39] P. Müller and A. Poetzsch-Heffter. Universes: A Type System for Alias and Dependency Control. Technical Report 279, Fernuniversität Hagen, 2001.

[40] J. Noble, J. Vitek, and J. Potter. Flexible Alias Protection. In *European Conference on Object Oriented Programming (ECOOP)*, 1998.

[41] N. Nystrom, S. Chong, and A. C. Myers. Scalable Extensibility via Nested Inheritance. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2004.

[42] J. Östlund and T. Wrigstad. Welterweight Java. In *International Conference on Objects, Components, Models and Patterns (TOOLS Europe)*, 2010.

[43] M. Parkinson. Class Invariants: The End of the Road? In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, 2007.

[44] C. Saito, A. Igarashi, and M. Viroli. Lightweight Family Polymorphism. *J. Funct. Program.*, 18(3), 2008.

[45] M. Smith. *A Model of Effects with an Application to Ownership Types*. PhD thesis, Imperial College, 2007.

[46] J. H. Spring, J. Privat, R. Guerraoui, and J. Vitek. StreamFlex: High-throughput Stream Programming in Java. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2007.

[47] M. Torgersen. The Expression Problem Revisited. In *European Conference on Object Oriented Programming (ECOOP)*, 2004.

[48] P. Wadler. The Expression Problem. Message to Java-Genericity mailing list, November 1998.

[49] T. Wrigstad and D. Clarke. Existential Owners for Ownership Types. *Journal of Object Technology*, 6(4), 2007.

[50] T. Wrigstad, F. Pizlo, F. Meawad, L. Zhao, and J. Vitek. Loci: Simple Thread-Locality for Java. In *European Conference on Object Oriented Programming (ECOOP)*, 2009.

[51] T. Zhao, J. Baker, J. Hunt, J. Noble, and J. Vitek. Implicit Ownership Types for Memory Management. *Sci. Comput. Program.*, 71(3), 2008.