# Existential Quantification for Variant Ownership

Nicholas Cameron* and Sophia Drossopoulou

Imperial College London
{ncameron, scd}@doc.ic.ac.uk

**Abstract.** Ownership types characterize the topology of objects in the heap, through a characterization of the context to which an object belongs. They have been used to support reasoning, memory management, concurrency, *etc.* Subtyping is traditionally invariant w.r.t. contexts, which has often proven inflexible in some situations. Recent work has introduced restricted forms of subtype variance and unknown context, but in a rather ad-hoc and restricted way.
We develop Jo∃, a calculus which supports parameterisation of types, as well as contexts, and allows variant subtyping of contexts based on existential quantification. Jo∃ is more expressive, general, and uniform than previous works which add variance to ownership languages. Our explicit use of existential types makes the connection to type-theoretic foundations from existential types more transparent. We prove type soundness for Jo∃ and extend it to $\text{Jo}\exists_{deep}$ which enforces the owners-as-dominators property.

## 1 Introduction

Ownership types [9,10,11] support a characterization of the topology of objects in the heap. They have been successfully applied in many areas. Boyapati [3] et al. annotated several Java library classes and multithreaded server programs, effectively preventing data races. Vitek et al. used ownership types to support memory management in real time systems, with applications such as flying unmanned aircraft [2], while Aldrich et al. used ownership to enforce software architectures in large, real-world software [1].

Usually ownership types are expressed by classes parameterised by formal *context parameters*, *e.g.,* `class C<o1,o2,o3> {...}`, and types parameterised by actuals, *e.g.,* `C<this,o2,o2>`. Context parameters represent objects. The first context parameter denotes the *owner* of the corresponding object. We say that an object is *inside* its owner, and all transitive owners of the latter. This implicitly defines a tree structure of owners in the heap.

*Deep ownership* systems enforce the *owners-as-dominators* property [11,9], which requires that the path to any object `o` from the root object passes through the owner of `o`. That is, objects are *dominated* by their owners. Such encapsulated objects are protected from direct and indirect access.

* Author's current address is Victoria University of Wellington, New Zealand.

In many variations of ownership types [9,10,22,23], actual context parameters must be *known* and invariant: they must not vary with execution or subtyping; *i.e.,* `C<o1>` is not a subtype of `C<o2>` even if `o1` is inside `o2`. This follows generic types: `List<Dog>` is not a subtype of `List<Animal>`.

Recent work on ownership types has introduced the concept of *unknown*, flexible contexts: universe types [13] support the annotation `any`, MOJO [6] uses the context parameter `?`, and effective ownership [19] uses an `any` context. These unknown owners introduce *variant subtyping*, whereby, *e.g.,* `C<o>` is a subtype of `C<?>`. Variant ownership types [18] support variance annotations to more precisely describe variance properties of ownership types.

All these systems are somewhat ad hoc in formalisation — there is no direct link to the underlying theory of existential types. In particular, they do not support:

**1** two or more context parameters are unknown, but known to be the same, *e.g.,* in the type ∃`o.C<o,o>`;
**2** context polymorphic methods [9,23] in the presence of variant contexts;
**3** upper *and* lower bounds on variant contexts;
**4** scoping of unknown contexts, *e.g.,* to distinguish a list of students which may have different owners, from a list of student which share the same unknown owners, *i.e.,* `List<this, ∃o.Student<o>>`, and ∃`o.List<this, Student<o>>`.

To bridge this gap, we develop Jo∃, which has its foundations in existential types and supports all these features. Jo∃ is a purely descriptive system, in that it only describes the heap topology, and guarantees that the topology is preserved, but does not restrict the topology in any way. We then develop a flavour of Jo∃, called Jo$\exists_{deep}$, which also supports deep ownership. We have distinguished deep ownership from the existential aspects, because descriptive ownership systems are useful in their own right (*e.g.,* to support reasoning with effects).

Jo∃ is a foundational, rather than usable, system. We expect it to be useful to reason about variance in ownership systems and to compare the various implementations of ownership variance. Whilst it is expressive and powerful, Jo∃ is verbose. Practical adoption of Jo∃ would require heavy syntactic sugaring.

Recent work with Java wildcards and similar systems [7,5,16,20] has used existential types to implement and formalise subtype variance in object-oriented languages. In these systems existential types are often implicit [20,16], a more programmer-friendly syntax obscures the underlying existential types. Packing and unpacking are usually implicit, even where quantification is explicit [5].

We use existential quantification of contexts to implement variant ownership. This solution is uniform and clearly related to its theoretical underpinnings; typing and the underlying mechanisms are refelcted in the syntax. Furthermore, in combination with type parameterisation, it is extremely expressive.

*Outline* In Sect. 2 we give an example explaining and motivating Jo∃. We present Jo∃ in Sect. 3 and Jo$\exists_{deep}$ in Sect. 4. We discuss these languages in Sect. 5 and their relation to related work in Sect. 6. We conclude in Sect. 7

## 2 Example

In this example we use a sugared syntax[1], rather than the verbose Jo∃ syntax, with implicit packing and unpacking of existential types. Such implicit packing and unpacking appears, for example, in Java wildcards; mapping from the sugared version to Jo∃ is simple [4]. We use `o→[a b]` to denote that the formal context parameter `o` has the lower bound `a` and upper bound `b`, that is, any instantiation of `o` must be inside `b` and outside[2] `a` in the ownership hierarchy.

```
class Worker<manager, company outside manager> {
    List<this, Worker<manager, company>> colleagues;
    ∃o→[⊥ company].List<this, Worker<o, company>> workGroup;
    ∃o→[manager company].Worker<o, company> mentor;

    void mixGroups() {
        workGroup = colleagues;
        //colleagues = workGroup;                          ERROR
        //colleagues.add(workGroup.get(0));                ERROR
        //workGroup.add(colleagues.get(0));                ERROR
    }
}

class Company extends Object<◯> {
    Worker<this, this> director;
    Worker<director, this> headOfMarketing;
    ∃o→[⊥ director].Worker<o, this> employeeOfTheMonth;
    List<this, ∃o→[⊥ this].Worker<o, this>> payroll;

    <m> void processColleagues(Worker<m, this> w) {
        for (Worker<m, this> c : w.colleagues) { ... }
    }

    void mentorEmpMonth() {
        employeeOfTheMonth.mentor = director;
        //employeeOfTheMonth.mentor =
        //    new Worker<headOfMarketing, this>;          ERROR
    }
}
```

Our example is part of a human resources system for a large company. Each worker in the company is owned by its `manager`; the employees form a hierarchy with the `director` at its root. In the `Worker` class, each worker keeps a list of his

[1] We also use fields as context parameters. This is not implemented in Jo∃, but is a relatively easy extension. It is present in, for example, MOJO [6]

[2] We say `o` outside `o′` to mean `o′` inside `o`.

`colleagues`. Each colleague is a `Worker` with the same `manager` as `this`. In the `Company` class, we store references to the `director` and the head of marketing, whose immediate `manager` is the `director`.

So far, we have only used features present in classical ownership types systems. We use existential types where the precise owner of objects is unknown and highlight the features listed in the previous section, *e.g.,* **1**. In the `Worker` class, `mentor` is some worker who either works with or indirectly manages that worker, but whose exact position in the management hierarchy is not specified (**3**). A worker may work with some other team of workers in the company (a team is assumed to have a single manager). For example, an engineer may have contact with the management team. This group (`workGroup`) may have any manager in the company, and this is represented by the existential type. Since we assume all members of the group have the same owner, the existential quantification is outside the `List` (**4**).

In the `Company` class, the `employeeOfTheMonth` may be any `Worker` in the company, her manager is not important. The `payroll` keeps track of every worker in the company. Each worker on the `payroll` may have a different manager.

The method `processColleagues` takes a worker (`w`) as a parameter and performs some action on each of his colleagues. Since the method is polymorphic in the manager (`m`) of `w`, we can name `m` as the owner of `w`'s colleagues, `c` (**2**).

In `mixGroups` we can set `workGroup` to `colleagues` because `manager` (the manager of `colleagues`) is within the bounds specified in the type of `workGroup`. We cannot set `colleagues` to `workGroup`, nor add an element of `colleagues` to `workGroup`, because `workGroup` may have any manager, not necessarily `this`. Even though we can set `workGroup` to `colleagues`, we cannot add an element of `colleagues` to `workGroup` because although the owner of the `workGroup` may be any owner, it is a specific owner and not necessarily `manager`[3].

*Owners-as-dominators.* Even in a deep ownership system it can be safe and desirable to support subtype variance. A `Worker` instance and his `mentor` (though not his `workGroup`) satisfy owners-as-dominators in Jo$\exists_{deep}$. `mentorEmpMonth` sets the `mentor` of the `employeeOfTheMonth` to the `director`. This preserves owners-as-dominators since the `director` must transitively manage (own) the `employeeOfTheMonth`, no matter who that is. Setting the `employeeOfTheMonth`'s `mentor` to a new worker owned by the `headOfMarketing` would violate owners-as-dominators and is not allowed. This is because the `employeeOfTheMonth` may not be transitively owned by this new worker.

## 3 Jo$\exists$

In this section we present the most interesting parts of Jo$\exists$, a minimal object-oriented language in the style of FGJ [15], with parametrisation of methods and

[3] here, the owner is `manager` due to the earlier assignment, but in general it will be unknown.

classes by context and type parameters, and existential quantification of contexts. In order to demonstrate ownership properties, we include field assignment and a mutable heap. Jo$\exists$ is fully described in the first author's PhD thesis [4] along with much extra detail that could not be included here for space reasons.

Subtype variance in Jo$\exists$ is implemented by existential quantification. Existential types are explicit and are introduced and eliminated (packed and unpacked) using `close` and `open` expressions. Thus, we follow the more traditional model of existential types [7], rather than the Java 5.0 approach of using implicit packing and unpacking.

Neither the ownership or existential quantification features of Jo$\exists$ interact with subclassing. Furthermore, the benefits of existential quantification in Jo$\exists$ do not depend on subclassing, nor the absence of subclassing. For these reasons, and because the standard solution to subclassing in ownership types systems is long known [10], we elide subclassing and inheritance. This simplifies the presentation of Jo$\exists$ and its proofs. Jo$\exists$ could be extended to include subclassing by extending the subtyping and method and field lookup rules following FGJ [15]. Subclassing must preserve the formal owner of an object [10]. There are no changes to any of the rules involving quantification.

We are primarily interested in type parameterisation to increase expressiveness of ownership types, rather than to investigate features of generic types. We therefore treat type parameterisation simply and do not support bounds on formal type parameters, nor existential quantification of type variables.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| `e` | ::= | `null` \| `x` \| $\gamma$`.f` \| $\gamma$`.f = e` \| $\gamma$`.<`$\overline{\texttt{a}}$`, `$\overline{\texttt{T}}$`>m(`$\overline{\texttt{e}}$`)` \| `new C<`$\overline{\texttt{a}}$`, `$\overline{\texttt{T}}$`>` \| $\overline{\texttt{open e as x,o in}}$ `e` \| `close e with` $\overline{\texttt{o}\rightarrow\texttt{[b b]}}$ `hiding` $\overline{\texttt{a}}$ \| $\iota$ \| `err` | | | | | | *expressions* |
| `Q` | ::= | `class C<`$\Delta$`, `$\overline{\texttt{X}}$`> {`$\overline{\texttt{T f;}}$ $\overline{\texttt{W}}$`}` | | | | | *class declarations* |
| `W` | ::= | `<`$\Delta$`, `$\overline{\texttt{X}}$`> T m(`$\overline{\texttt{T x}}$`) {return e;}` | | | | | *method declarations* |
| `v` | ::= | $\overline{\texttt{close v with o}\rightarrow\texttt{[b b]}}$ `hiding` $\overline{\texttt{r}}$ \| $\iota$ \| `null` \| `err` | | | | | *values* |
| `N` | ::= | `C<`$\overline{\texttt{a}}$`, `$\overline{\texttt{T}}$`>` | *class types* | `a` | ::= | `o` \| `x` \| $\bigcirc$ \| $\iota$ | *contexts* |
| `R` | ::= | `C<`$\overline{\texttt{r}}$`, `$\overline{\texttt{T}}$`>` | *runtime types* | `r` | ::= | $\bigcirc$ \| $\iota$ | *runtime contexts* |
| `M` | ::= | `N` \| `X` | *non-existential types* | `b` | ::= | `a` \| $\bot$ | *bounds* |
| `T` | ::= | `M` \| $\exists\Delta$`.N` | *types* | | | | |
| $\Psi$ | ::= | $\overline{\texttt{X}\rightarrow\texttt{[b}_l\ \texttt{b}_u\texttt{]}}$ | *type environments* | `x, y` | | | *variables* |
| $\Delta$ | ::= | $\overline{\texttt{o}\rightarrow\texttt{[b}_l\ \texttt{b}_u\texttt{]}}$ | *context environments* | `X, Y` | | | *type variables* |
| $\gamma$ | ::= | `x` \| $\iota$ \| `null` | *vars and addresses* | `o` | | | *formal owners* |
| $\Gamma$ | ::= | $\overline{\gamma\texttt{:T}}$ | *var environments* | `C` | | | *classes* |
| $\mathcal{H}$ | ::= | $\overline{\iota\rightarrow\texttt{\{R; }\overline{\texttt{f}\rightarrow\texttt{v}}\texttt{\}}}$ | *heaps* | $\iota$ | | | *addresses* |

**Fig. 1.** Syntax of Jo$\exists$.

*Syntax.* The syntax of Jo∃ is given in Fig. 1. Entities only used at runtime are in grey. Jo∃ includes expressions for accessing variables (`x`, which includes `this`) and addresses ($\iota$), object creation, `null` (for field initialisation), field access and assignment, method invocation, and packing and unpacking of existential types.

Class and method declarations (`Q` and `W`) are parameterised by context (`o`) and type (`X`) parameters. The former have upper and lower bounds (bounds are actual context parameters — not subtype bounds — and limit the bounded formal context to some part of the ownership hierarchy), and so methods and classes are considered to be parameterised by *context environments* ($\Delta$). These are mappings from formal context parameters to their bounds (`o`$\rightarrow$`[`$\mathtt{b}_l$ $\mathtt{b}_u$`]`).

Contexts (`a`) consist of context variables (`o`), variables (`x`) and the *world context* (the root object), $\bigcirc$. At runtime we may also use addresses. Runtime contexts (`r`) are restricted to addresses and $\bigcirc$.

Variable environments, $\Gamma$, map variables to their types. Type environments, $\Psi$, map type variables to bounds on a context. Type variables do not have bounds on the types they may take. The bounds contained in $\Psi$ define upper and lower bounds on the owner of actual types. If the lower and upper bounds on the owner of `X` are $\mathtt{b}_l$ and $\mathtt{b}_u$, then for `C<o>` to instantiate `X`, `o` must be outside $\mathtt{b}_l$ and inside $\mathtt{b}_u$. The bounds in $\Psi$ are manufactured by the type system (in T-Class in Fig. 4 and T-Method [4]) and cannot be defined by the programmer. In Jo∃ and Jo$\exists_{deep}$, upper bounds in $\Psi$ are always $\bigcirc$ and, in effect, are never used; however, we keep upper bounds to allow for easy extension. We only use the lower bound to support deep ownership in Jo$\exists_{deep}$ (Sect. 4).

To model execution we use a heap, $\mathcal{H}$, which maps addresses ($\iota$) to records representing objects. Each record contains the type of the object and a mapping from field names to values. Values (`v`) are addresses or `close` expressions that pack addresses.

*Types in Jo∃.* The syntax of types in Jo∃ is given in Fig. 1. Class types (`N`) are class names parameterised by actual type and context parameters. The first context parameter is the owner of objects with that type. Class types may be existentially quantified by a context environment to give existential types. For example, `∃o.List<o, Animal>` denotes a list owned by *some* owner. For conciseness in examples, we omit bounds and empty parameter lists where convenient.

By combining existential quantification with type parameterisation we can express many interesting and useful types: `∃o.List<o, Animal<this>>` denotes a list owned by some unknown owner where each element is an `Animal` owned by `this`, while `∃o1,o2.List<o1, Animal<o2>>` denotes a list owned by some owner where all elements are owned by the same owner which may be different from the owner of the list, and `∃o1.List<o1, ∃o2.Animal<o2>>` denotes a list where each element is owned by some owner and the owner of each element may be different, finally, `∃o.List<o, Animal<o>>` denotes a list where each element in the list and the list itself are owned by the same, unknown, owner.

*Subtyping and the Inside Relation.* The inside relation relates contexts and is defined by the rules given in Fig. 2. We say that $\mathtt{o}_1$ is inside $\mathtt{o}_2$ ($\Delta; \Gamma \vdash \mathtt{o}_1 \preceq \mathtt{o}_2$),

$$\frac{}{\Delta;\Gamma \vdash \mathtt{M} <: \mathtt{M}}\ \text{(S-Reflex)} \qquad \frac{\Delta;\Gamma \vdash \overline{\mathtt{b}_u \preceq \mathtt{b}'_u} \qquad \Delta;\Gamma \vdash \overline{\mathtt{b}'_l \preceq \mathtt{b}_l}}{\Delta;\Gamma \vdash \exists\overline{\mathtt{o} \to [\mathtt{b}_l\ \ \mathtt{b}_u]}.\mathtt{N} <: \exists\overline{\mathtt{o} \to [\mathtt{b}'_l\ \ \mathtt{b}'_u]}.\mathtt{N}}\ \text{(S-Full)}$$

$$\frac{}{\Delta;\Gamma \vdash \mathtt{b} \preceq \mathtt{b}}\ \text{(I-Reflex)} \qquad \frac{\Delta;\Gamma \vdash \mathtt{b} \preceq \mathtt{b}'' \qquad \Delta;\Gamma \vdash \mathtt{b}'' \preceq \mathtt{b}'}{\Delta;\Gamma \vdash \mathtt{b} \preceq \mathtt{b}'}\ \text{(I-Trans)} \qquad \frac{\Delta;\Gamma \vdash \mathtt{b}\ \text{OK}}{\Delta;\Gamma \vdash \mathtt{b} \preceq \bigcirc}\ \text{(I-World)}$$

$$\frac{\Delta;\Gamma \vdash \mathtt{b}\ \text{OK}}{\Delta;\Gamma \vdash \bot \preceq \mathtt{b}}\ \text{(I-Bottom)} \qquad \frac{\Gamma(\gamma) = \mathtt{C<\overline{a},\ \overline{T}>}}{\Delta;\Gamma \vdash \gamma \preceq \mathtt{a}_0}\ \text{(I-Owner)} \qquad \frac{\Delta(\mathtt{o}) = [\mathtt{b}_l\ \ \mathtt{b}_u]}{\Delta;\Gamma \vdash \mathtt{o} \preceq \mathtt{b}_u \quad \Delta;\Gamma \vdash \mathtt{b}_l \preceq \mathtt{o}}\ \text{(I-Bound)}$$

**Fig. 2.** Jo∃ subtyping, and the inside relation for owners and environments.

if $\mathtt{o}_1$ is transitively owned by $\mathtt{o}_2$. The inside relation is reflexive, transitive, and has top and bottom elements — the world and bottom contexts, respectively. I-Owner asserts that every variable and address is inside the declared owner of its type (if its type is a class type). For example, if `this` has type `C<o>`, then `this` is inside `o`. I-Bound gives that a formal context is within its bounds.

Subtyping is also given in Fig. 2. Since there is no subclassing in Jo∃, subtyping of non-existential types is given only by reflexivity. Subtyping between existential types follows the full variant of existential subtyping [14,7]. Existential types are subtypes where the bounds of quantified contexts in the subtype are more strict than in the supertype.

$$\frac{\mathtt{o} \in dom(\Delta)}{\Delta;\Gamma \vdash \mathtt{o}\ \text{OK}}\ \text{(F-Owner)} \qquad \frac{}{\Delta;\Gamma \vdash \bigcirc\ \text{OK}}\ \text{(F-World)} \qquad \frac{}{\Delta;\Gamma \vdash \bot\ \ \text{OK}}\ \text{(F-Bottom)} \qquad \frac{\Gamma(\gamma) = \mathtt{N}}{\Delta;\Gamma \vdash \gamma\ \text{OK}}\ \text{(F-Var)}$$

$$\frac{\begin{array}{c}\mathtt{class\ C<\overline{o \to [b_l\ \ b_u]},\ \overline{X}>...} \qquad \Delta;\Gamma \vdash \overline{\mathtt{a}}\ \text{OK} \\ \Delta;\Gamma,\mathtt{this:C<\overline{a},\ \overline{X}>} \vdash \overline{[\overline{a}/\overline{o}]\mathtt{b}_l \preceq \mathtt{a}} \qquad \Delta;\Gamma,\mathtt{this:C<\overline{a},\ \overline{X}>} \vdash \mathtt{a} \preceq \overline{[\overline{a}/\overline{o}]\mathtt{b}_u} \\ \Psi;\Delta;\Gamma \vdash \overline{\mathtt{T}}\ \text{OK} \qquad |\overline{\mathtt{T}}| = |\overline{\mathtt{X}}|\end{array}}{\Psi;\Delta;\Gamma \vdash \mathtt{C<\overline{a},\ \overline{T}>}\ \text{OK}}\ \text{(F-Class)}$$

$$\frac{\mathtt{X} \in dom(\Psi)}{\Psi;\Delta;\Gamma \vdash \mathtt{X}\ \text{OK}}\ \text{(F-Type-Var)} \qquad \frac{\Delta;\Gamma \vdash \overline{\mathtt{o} \to [\mathtt{b}_l\ \ \mathtt{b}_u]}\ \text{OK} \qquad \Psi;\Delta,\overline{\mathtt{o} \to [\mathtt{b}_l\ \ \mathtt{b}_u]};\Gamma \vdash \mathtt{N}\ \text{OK}}{\Psi;\Delta;\Gamma \vdash \exists\overline{\mathtt{o} \to [\mathtt{b}_l\ \ \mathtt{b}_u]}.\mathtt{N}\ \text{OK}}\ \text{(F-Exist)}$$

**Fig. 3.** Jo∃ well-formed contexts and types.

*Well-formedness.* Well-formed contexts and types are given in Fig. 3. An owner variable is well-formed if it has class type; this guarantees precise information

about all unquantified contexts, and that the set of contexts is closed under substitution. This restriction abides by the philosophy of existential types, that abstract packages must be unpacked to be used.

Well-formed class types (F-Class) require the class name to have been declared, actual context parameters to be within the bounds of formal context parameters, the number of actual type parameters to match the number of formal type parameters, and actual context and type parameters to be well-formed. Well-formed environments (used in F-Exists) are elided, the only interesting aspect is that we require the lower bound of each context variable to be inside its corresponding upper bound.

To check that actual context parameters are within their corresponding bounds, the judging environments are extended with `this` mapped to `C<`$\overline{\texttt{a}}$`, `$\overline{\texttt{X}}$`>`, i.e., the class type with actual context parameters and formal type parameters. This is necessary because $\overline{\texttt{b}_l}$ and $\overline{\texttt{b}_u}$ may involve `this`. We cannot substitute for `this`, because there may not be a variable or address that contains the object to be substituted. We use a mixture of actual context parameters ($\overline{\texttt{a}}$) and formal type parameters ($\overline{\texttt{X}}$) because of the order of application of substitution lemmas in the proofs. Using $\overline{\texttt{X}}$ is safe, even though $\overline{\texttt{X}}$ are not in scope, because the type parameters of types are never used in the rules defining the inside relation.

$$\frac{\Psi;\Delta;\Gamma \vdash \gamma : \texttt{N} \qquad fType(\texttt{f},\gamma,\texttt{N}) = \texttt{T}}{\Psi;\Delta;\Gamma \vdash \gamma\texttt{.f} : \texttt{T}}\ \text{(T-Field)}$$

$$\frac{\Psi;\Delta;\Gamma \vdash \gamma : \texttt{N} \qquad fType(\texttt{f},\gamma,\texttt{N}) = \texttt{T} \qquad \Psi;\Delta;\Gamma \vdash \texttt{e} : \texttt{T}}{\Psi;\Delta;\Gamma \vdash \gamma\texttt{.f = e} : \texttt{T}}\ \text{(T-Assign)}$$

$$\frac{\Psi;\Delta;\Gamma \vdash \texttt{C<}\overline{\texttt{a}}\texttt{, }\overline{\texttt{U}}\texttt{>}\ \text{OK}}{\Psi;\Delta;\Gamma \vdash \texttt{new C<}\overline{\texttt{a}}\texttt{, }\overline{\texttt{U}}\texttt{>} : \texttt{C<}\overline{\texttt{a}}\texttt{, }\overline{\texttt{U}}\texttt{>}}\ \text{(T-New)}$$

$$\frac{\Psi;\Delta;\Gamma \vdash \gamma : \texttt{N} \qquad \Psi;\Delta;\Gamma \vdash \overline{\texttt{e : T}} \qquad \Delta;\Gamma \vdash \overline{\texttt{a}}\ \text{OK} \qquad \Psi;\Delta;\Gamma \vdash \overline{\texttt{U}}\ \text{OK} \qquad mType_{\Delta;\Gamma}(\texttt{m<}\overline{\texttt{a}}\texttt{, }\gamma\texttt{, }\overline{\texttt{U}}\texttt{>},\texttt{N}) = \overline{\texttt{T}}\texttt{→T}}{\Psi;\Delta;\Gamma \vdash \gamma\texttt{.<}\overline{\texttt{a}}\texttt{, }\overline{\texttt{U}}\texttt{>m(}\overline{\texttt{e}}\texttt{)} : \texttt{T}}\ \text{(T-Invk)}$$

$$\frac{\Psi;\Delta;\Gamma \vdash \texttt{e} : \exists\overline{\texttt{o}\rightarrow\texttt{[b}_l\texttt{ b}_u\texttt{]}}\texttt{.N} \qquad \Psi;\Delta,\overline{\texttt{o}\rightarrow\texttt{[b}_l\texttt{ b}_u\texttt{]}};\Gamma,\texttt{x:N} \vdash \texttt{e}' : \texttt{T} \qquad \Psi;\Delta;\Gamma \vdash \texttt{T}\ \text{OK}}{\Psi;\Delta;\Gamma \vdash \texttt{open e as x,}\overline{\texttt{o}}\texttt{ in e}' : \texttt{T}}\ \text{(T-Open)}$$

$$\frac{\Delta;\Gamma \vdash \overline{\texttt{[}\overline{\texttt{a/o}}\texttt{]b}_l \preceq \texttt{a}} \qquad \Delta;\Gamma \vdash \overline{\texttt{a} \preceq \texttt{[}\overline{\texttt{a/o}}\texttt{]b}_u} \qquad \Delta;\Gamma \vdash \overline{\texttt{a}}\ \text{OK} \qquad \Psi;\Delta;\Gamma \vdash \texttt{e} : \texttt{[}\overline{\texttt{a/o}}\texttt{]N} \qquad \Psi;\Delta;\Gamma \vdash \exists\overline{\texttt{o}\rightarrow\texttt{[b}_l\texttt{ b}_u\texttt{]}}\texttt{.N}\ \text{OK}}{\Psi;\Delta;\Gamma \vdash \texttt{close e with }\overline{\texttt{o}\rightarrow\texttt{[b}_l\texttt{ b}_u\texttt{]}}\texttt{ hiding }\overline{\texttt{a}} : \exists\overline{\texttt{o}\rightarrow\texttt{[b}_l\texttt{ b}_u\texttt{]}}\texttt{.N}}\ \text{(T-Close)}$$

$$\frac{\Psi = \overline{\texttt{X}\rightarrow\texttt{[}\bot\bigcirc\texttt{]}} \qquad \emptyset;\texttt{this:C<}\overline{\texttt{o}}\texttt{, }\overline{\texttt{X}}\texttt{>} \vdash \overline{\texttt{o}\rightarrow\texttt{[b}_l\texttt{ b}_u\texttt{]}}\ \text{OK} \qquad \Psi;\overline{\texttt{o}\rightarrow\texttt{[b}_l\texttt{ b}_u\texttt{]}};\texttt{this:C<}\overline{\texttt{o}}\texttt{, }\overline{\texttt{X}}\texttt{>} \vdash \overline{\texttt{W}}\texttt{, }\overline{\texttt{T}}\ \text{OK}}{\vdash \texttt{class C<}\overline{\texttt{o}\rightarrow\texttt{[b}_l\texttt{ b}_u\texttt{]}}\texttt{, }\overline{\texttt{X}}\texttt{> \{}\overline{\texttt{T f; }}\overline{\texttt{W}}\texttt{\}}\ \text{OK}}\ \text{(T-Class)}$$

**Fig. 4.** Jo∃ expression and class typing rules.

*Typing.* Type rules are given in Fig. 4. Field and variable access (T-Field and T-Var) are close to those of FGJ [15]. Field assignment (T-Assign) is a straightforward extension of field access. We adopt the standard subsumption rule (T-Sub). In object creation (T-New), we create uninitialised objects, we do not support constructors. T-Null allows `null` to take any well-formed type. Method invocation is also close to FGJ, with the addition that actual context parameters must be well-formed and within their corresponding formal bounds.

In T-Field, T-Assign, and T-Invk, the receiver is restricted to $\gamma$. This allows us to substitute $\gamma$ for `this` in field and method types without requiring dependent typing. Expressivity is not lost since the programmer can use an `open` expression with empty $\overline{\mathtt{o}}$ to act as a let expression.

To type check `open` and `close` expressions we follow Fun [8] and other classical existential types systems. The type of expression `e` is unpacked to an owner environment, $\overline{\mathtt{o}\rightarrow[\mathtt{b}_l\ \mathtt{b}_u]}$, and unquantified type, `N`. We then judge the body of open ($\mathtt{e}'$) by extending $\Delta$ with $\overline{\mathtt{o}\rightarrow[\mathtt{b}_l\ \mathtt{b}_u]}$ and adding a fresh variable, `x`, with type `N` to $\Gamma$; `x` stands for the unpacked value of `e`. We ensure no variables escape the scope of the `open` expression by checking that the result type, `T`, is well-formed without $\overline{\mathtt{o}}$ or `x`.

The `close` expression packs an expression `e` by hiding some of the context parameters present in `e`'s type. For example, if `e` has type `C<this>`, then the expression `close e with o hiding this` has the existential type ∃`o`.`C<o>`.

*Example.* The assignment `employeeOfTheMonth.mentor = director` from the example in Sect. 2 is represented with explicit packing and unpacking as:

```
open employeeOfTheMonth as e,m in
    e.mentor = close director with o→[m this] hiding this;
```

Under an environment where `e` has type `Worker<m, this>`, the close and assignment expressions have type ∃`o`→`[m this].Worker<o, this>` by T-Close, and by T-Assign and S-Reflex, respectively. By T-Subs, S-Full, and I-Bttm, the assignment has the `m`-free type ∃`o`→`[⊥ this].Worker<o, this>`. `employeeOfTheMonth` (of type ∃`m`→`[⊥ director].Worker<m, this>`) can be unpacked as `e` (of type `Worker<m, this>`), used in type checking the assignment, and therefore T-Open can be applied, giving the entire expression the type ∃`o`→`[⊥ this].Worker<o, this>`.

*Dynamic Semantics.* We elide most of the operational semantics of Jo∃, they are mostly standard[4]. Reduction of open and close expressions is given by the following rule, taken from the classical formulations of existential types [21]:

$$\mathtt{open\ (close\ v\ with\ }\overline{\mathtt{o}\rightarrow[\mathtt{b}_l\ \mathtt{b}_u]}\mathtt{\ hiding\ }\overline{\mathtt{r}}\mathtt{)\ as\ x,}\overline{\mathtt{o}}\mathtt{\ in\ e};\mathcal{H}\rightsquigarrow[\overline{\mathtt{r}/\mathtt{o}},\mathtt{v/x}]\mathtt{e};\mathcal{H}$$

The `open` and `close` sub-expressions are eliminated, leaving the body of `open` (`e`) with formal variables replaced by the packed value and hidden contexts. For example, `open (close 3 with o hiding 2) as x,o in (this.<o>m(x))`, where `2` and `3` are addresses, reduces to `this.<2>m(3)` (we replace `x` by `3` and `o` by `2`).

[4] Object creation, performed in R-New, creates a new object with all its fields set to `null`; i.e., we do not support constructors.

$$\frac{\begin{array}{c}\forall \iota \to \{\texttt{C<}\overline{\texttt{r}}\texttt{, }\overline{\texttt{T}}\texttt{>;}\overline{\texttt{f}\to\texttt{v}}\} \in \mathcal{H} : \\ \emptyset; \Delta; \mathcal{H} \vdash \texttt{C<}\overline{\texttt{r}}\texttt{, }\overline{\texttt{T}}\texttt{>}\ \text{OK} \\ fType(\texttt{f}, \iota, \texttt{C<}\overline{\texttt{r}}\texttt{, }\overline{\texttt{T}}\texttt{>}) = \texttt{T}' \qquad \emptyset; \Delta; \mathcal{H} \vdash \overline{\texttt{v} : \texttt{T}'} \\ \forall \texttt{v} \in \overline{\texttt{v}} : add(\texttt{v})\ \textit{defined} \Rightarrow add(\texttt{v}) \in dom(\mathcal{H})\end{array}}{\Delta \vdash \mathcal{H}\ \text{OK}}\ \text{(F-Heap)}$$

$$\frac{\begin{array}{c}\Delta \vdash \mathcal{H}\ \text{OK} \\ \forall \iota \in fv(\texttt{e}) : \iota \in dom(\mathcal{H})\end{array}}{\Delta; \mathcal{H} \vdash \texttt{e}\ \text{OK}}\ \text{(F-Config)}$$

**Fig. 5.** Jo∃ well-formed heaps and configurations.

In Fig. 5 we give the definitions of well-formed heaps and configurations. Most premises are standard. We insist that the address of all referenced values are in the domain of the heap. The address of a value is given by the partial function *add*, defined as:

$$add(\texttt{v}) = \begin{cases} \iota, & \text{if } \texttt{v} = \iota \\ add(\texttt{v}'), & \text{if } \texttt{v} = \texttt{close v}'\texttt{...} \\ \textit{undefined}, & \text{otherwise} \end{cases}$$

which recursively unwraps abstract packages, returning the address within. Thus, $add(v)$ is defined if v is neither `null` nor `null` wrapped in a `close` expression.

*Type Soundness.* Type soundness in Jo∃ guarantees that the types of variables accurately reflect their contents, including ownership information. Furthermore, the ownership hierarchy defined statically in a program describes the heap when that program is executed. Although these properties do not constitute an encapsulation property, they are necessary when using ownership information to reason about programs, for example using effects [10]. We show type soundness for Jo∃ by proving progress and preservation (subject reduction):

**Theorem (progress)** *For any* $\mathcal{H}, \texttt{e}, \texttt{T}$*, if* $\emptyset; \emptyset; \mathcal{H} \vdash \texttt{e} : \texttt{T}$ *and* $\emptyset \vdash \mathcal{H}$ OK *then either there exists* $\mathcal{H}', \texttt{e}'$ *such that* $\texttt{e}; \mathcal{H} \rightsquigarrow \texttt{e}'; \mathcal{H}'$ *or there exists* v *such that* $\texttt{e} = \texttt{v}$*.*

**Theorem (subject reduction)** *For any* $\Delta, \mathcal{H}, \mathcal{H}', \texttt{e}, \texttt{e}', \texttt{T}$*, if* $\emptyset; \Delta; \mathcal{H} \vdash \texttt{e} : \texttt{T}$ *and* $\texttt{e}; \mathcal{H} \rightsquigarrow \texttt{e}'; \mathcal{H}'$ *and* $\Delta; \mathcal{H} \vdash \texttt{e}$ OK *and* $\emptyset; \mathcal{H} \vdash \Delta$ OK *and* $\texttt{e}' \neq \texttt{err}$ *then* $\emptyset; \Delta; \mathcal{H}' \vdash \texttt{e}' : \texttt{T}$ *and* $\Delta; \mathcal{H}' \vdash \texttt{e}'$ OK*.*

Proofs are given in [4] and can be downloaded from:

http://www.doc.ic.ac.uk/~ncameron/papers/cameron_esop09_proofs.pdf

## 4 Jo∃$_{deep}$

Jo∃$_{deep}$ enforces the owners-as-dominators property. It differs from Jo∃ only in its definition of well-formed types, heaps, and classes. We define auxiliary functions to find the owner of an object in the heap ($own_{\mathcal{H}}(\texttt{v})$) and the owner of objects with type T ($own_{\Psi}(\texttt{T})$) in Fig. 6.

The owner of objects of type X is the lower bound on the owner of X, recorded in $\Psi$. To find the owner of objects with existential type ($\exists \Delta.\texttt{C<}\overline{\texttt{a}}\texttt{, }\overline{\texttt{T}}\texttt{>}$), we must find a context that is not quantified and that is inside the declared owner of the

$$\frac{}{own_\Psi(\texttt{C<}\overline{\texttt{a}}\texttt{, }\overline{\texttt{T}}\texttt{>}) = \texttt{a}_0} \qquad \frac{\Psi(\texttt{X}) = \texttt{[b}_l\ \texttt{b}_u\texttt{]}}{own_\Psi(\texttt{X}) = \texttt{b}_l} \qquad \frac{}{own_\Psi(\exists \Delta.\texttt{C<}\overline{\texttt{a}}\texttt{, }\overline{\texttt{T}}\texttt{>}) = glb_\Delta(\texttt{a}_0)}$$

$$\frac{\texttt{b} \notin dom(\Delta)}{glb_\Delta(\texttt{b}) = \texttt{b}} \qquad \frac{\Delta(\texttt{o}) = \texttt{[b}_l\ \texttt{b}_u\texttt{]}}{glb_\Delta(\texttt{o}) = glb_\Delta(\texttt{b}_l)}$$

$$\frac{\mathcal{H}(\iota) = \{\texttt{C<}\overline{\texttt{r}}\texttt{, }\overline{\texttt{T}}\texttt{> } \ldots\}}{own_\mathcal{H}(\iota) = \texttt{r}_0} \qquad \frac{}{own_\mathcal{H}(\texttt{close v with } \overline{\texttt{o}\rightarrow\texttt{[b}_l\ \texttt{b}_u\texttt{]}}\ \texttt{hiding } \overline{\texttt{r}}) = own_\mathcal{H}(\texttt{v})}$$

**Fig. 6.** Owner lookup functions for Jo$\exists_{deep}$.

type ($\texttt{a}_0$). This is accomplished by the *glb* function; $glb_\Delta(\texttt{b})$ finds the outermost object that is inside $\texttt{b}$ and not in the domain of $\Delta$.

The owners-as-dominators property manifests itself as an extra constraint on well-formed heaps; thus, we extend F-Heap (Fig. 5) as follows:

$$\frac{\begin{array}{ccc}\ldots & & \\ \forall \iota \in \mathcal{H} & \forall \texttt{v} \in \mathcal{H}(\iota) & \Delta; \mathcal{H} \vdash \iota \preceq own_\mathcal{H}(\texttt{v})\end{array}}{\Delta \vdash \mathcal{H}\ \text{OK}}$$

(F-Heap)

Similarly, Jo$\exists_{deep}$ requires some modifications to the well-formedness rules for class types and classes of Jo$\exists$:

$$\frac{\begin{array}{c}\ldots \\ \forall \texttt{a}_i \in \overline{\texttt{a}} : \Delta; \Gamma \vdash \texttt{a}_0 \preceq \texttt{a}_i \\ \forall \texttt{T}_i \in \overline{\texttt{T}} : \Delta; \Gamma \vdash \texttt{a}_0 \preceq own_\Psi(\texttt{T}_i)\end{array}}{\Psi; \Delta; \Gamma \vdash \texttt{C<}\overline{\texttt{a}}\texttt{, }\overline{\texttt{T}}\texttt{>}\ \text{OK}} \qquad \frac{\begin{array}{c}\ldots \\ \Psi = \overline{\texttt{X}\rightarrow\texttt{[o}_0\ \bigcirc\texttt{]}} \\ \bot \notin \overline{\texttt{T}}, \overline{\texttt{o}\rightarrow\texttt{[b}_l\ \texttt{b}_u\texttt{]}}\end{array}}{\vdash \texttt{class C<}\overline{\texttt{o}\rightarrow\texttt{[b}_l\ \texttt{b}_u\texttt{]}}\texttt{, }\overline{\texttt{X}}\texttt{> \{}\overline{\texttt{T f;}}\ \overline{\texttt{W}}\texttt{\}}\ \text{OK}}$$

(F-Class) (T-Class)

The extra premises in F-Class (together with the well-formedness rules for contexts) ensure that only contexts that are outside an object can be formed by substitution of actual for formal parameters in its class. The owner of an object ($\texttt{a}_0$) is, by definition, outside that object. The first extra premise ensures that the actual context parameters are outside $\texttt{a}_0$. The second premise ensures that the owners of any actual type parameters are outside $\texttt{a}_0$. Therefore, all types formed by substitution of contexts or types will have an owner outside `this`.

In T-Class we change the way $\Psi$ is created; the lower bounds in $\Psi$ are the formal owner of the class rather than $\bot$. This is required because of the changes we made to F-Class. The class declaration `class C<o, X> { C<o, X> f; }` would not type check without this change: otherwise `C<o, X>` would not be well-formed because $own_\Psi(\texttt{X})$ could not be derived to be outside `o`.

The second extra premise in T-Class requires that $\bot$ cannot appear as a bound in the formal context parameters of the class, nor in any existential types given to fields in the class. The intention is to ensure that the owner of all objects referenced by objects of the class (including the hidden owner of objects with existential type) is outside the referring object. Therefore, in the example in Sect. 2, the declaration of `workGroup` would be illegal in Jo$\exists_{deep}$.

We state the owners-as-dominators property in Jo$\exists_{deep}$ as:

**Theorem (Owners-as-dominators)** *For any* $\mathcal{H}$*, if* $\Delta \vdash \mathcal{H}$ OK *then* $\forall \iota \to \{\mathtt{R}; \{\overline{\mathtt{f} \to \mathtt{v}}\}\} \in \mathcal{H}, \forall \mathtt{v}_i \in \overline{\mathtt{v}} : \Delta; \mathcal{H} \vdash \iota \preceq own_{\mathcal{H}}(\mathtt{v}_i)$

This is given by the added premise to F-HEAP; we prove that this is maintained under execution as part of the proof of subject-reduction [4].

## 5 Discussion

The expressivity of types in Jo∃ comes from the combination of existential quantification of contexts and type parameterisation. The formalisation of Jo∃ follows from these starting points and the decision to use explicit packing and unpacking, which simplifies the type rules and proofs for Jo∃. The natural and uniform emergence of the calculus is reassuring.

Allowing packed values to be be values (and thus stored in the heap) follows earlier work [8,21,14] on existential types and is a natural consequence of explicit packing. However, the owners-as-dominators property is usually phrased assuming that all values are objects (addresses in Jo∃). We must therefore consider how to describe owners-as-dominators in the presence of packed values. We do this by not distinguishing between packed values and the objects that they abstract. This ensures that existential quantification cannot hide violations of owners-as-dominators.

In the type system of $\text{Jo}\exists_{deep}$, we had to extend the usual restrictions found in ownership systems to enforce owners-as-dominators. Requiring context parameters to be outside an object's owner is standard, we needed to extend this to deal with quantified context variables and type parameters. The crucial observation is that, in enforcing owners-as-dominators, we always wish to show that a value is outside the object that refers to it. It is therefore conservative to use a lower bound on a value's owner rather than the value's owner itself. The additional premises in F-CLASS of $\text{Jo}\exists_{deep}$ can thus deal with lower bounds on parameters. In the case of quantified context parameters this means that we can use their greatest lower bound. For type parameters we use the lower bound stored in $\Psi$; this motivates using $\Psi$ in Jo∃ rather than just a set of type variables.

## 6 Related Work

*Generics and Ownership Types.* Type and ownership information in ownership types systems is usually kept separate [9,12,25], as in Jo∃. Surprisingly, in OGJ [22], these two kinds of parameters can be expressed using only type parameters. This leads to a small and uniform extension of generic Java that implements deep ownership. The fact that context parameterisation can be encoded using type extension highlights the similarity of the two systems. It will be interesting future work to add Jo∃'s existential types to OGJ and, it is hoped, reap the benefits of Jo∃ in a more realistic language.

*Existential Types.* Existential quantification of ownership domains in *System* $F_{own}$ [17] allows domains to be passed around even if they cannot be named. System $F_{own}$ supports existential quantification of types, absent in Jo∃, but does not support subtyping and so existential quantification does not lead to variance.

*Infinitary ownership types* [9] use existential types to abstract contexts which cannot be named. Because of dynamically created contexts, this is necessary to avoid dependent typing. Existential types in Jo∃ can be used in the same way. However, since contexts cannot be dynamically created, abstraction is not necessary to avoid dependent typing.

Existential owners can be used in dynamic casts [24]. Casts are not supported in Jo∃, but they should be straightforward to add. Existential downcasting could then be encoded in Jo∃ by casting using an existential type.

*Variance. Variant ownership types* [18] are a programmer friendly way to support use-site subtype variance, and have very similar behaviour to existential types. Jo∃ types are more expressive as they allow lower and upper bounds on contexts (as opposed to upper *or* lower bounds), type parameters, and explicit quantification (to express types such as ∃`o.C<o, o>`).

*MOJO* [6] uses `?` to denote an unknown context parameter. This corresponds to an existentially quantified context bounded by ⊥ and ◯ in Jo∃. In MOJO, `?` may be used as an actual context parameter.

In the case of field access, substitution of `?` (not found in other systems such as Wild FJ [20]) produces a similar behaviour to existential types in Jo∃. To prevent field assignment and method call where `?` would appear as a type parameter by substitution (but not where `?` is written in the type), *strict* method and field lookup are used. Likewise in Jo∃, field assignment or method call where the receiver has existential type is type incorrect. Variant types in MOJO are, therefore, treated in the same way as unbounded existential types in Jo∃.

*Universes* [13] support limited subtype variance through the `any` notation. Universe types can be given corresponding types in Jo∃: `any C` corresponds to ∃`o`→`[`⊥◯`].C<o>`, `peer C` corresponds to `C<o>` (where `o` is the owner of the class declaration in which the type appears), and `rep C` corresponds to `C<this>`. The viewpoint adaptation[5] rules of universes correspond to substitution of owners and unpacking and packing in Jo∃. Generic universes [12] can be described using this correspondence and Jo∃'s type parameterisation.

An `any` context is used to facilitate variance in *effective ownership* [19]. During field and method type lookup, all substitutions of `any` for `x` are replaced with substitutions of `unknown` for `x`. This mechanism is similar to the abstract contexts of variant ownership types [18] and `?` in MOJO. Similarly to these systems, it should be possible to encode the ownership structure of effective ownership in Jo∃. Effective owners (per-method owners) are currently beyond the scope of Jo∃. An effective owner cannot be `any`, and so there is no variance aspect to these owners.

[5] Viewpoint adaptation is the change in universe annotations when considering a type in a different context from the one in which it was declared.

In most related work [6,13,18], the treatment of unknown contexts is specific to the underlying system; our approach is founded in the theory of existential types and makes clear the relationship between variant types and their behaviour. We discuss in more detail how Jo∃ can be used to encode and compare the systems described in this section in [4].

## 7 Conclusion and Future Work

Jo∃ supports context variance in a uniform and transparent fashion using existential types. Expressivity is improved by combining existential quantification of contexts with type parameterisation. We have extended Jo∃ to support owners-as-dominators and proved both versions sound.

Jo∃ can be used to compare and encode ownership systems with different kinds of variance or existential types. Existing mechanisms for supporting context variance have the same behaviour as existential types in Jo∃ and can be easily encoded (even if other language features cannot). Explicit existential types can give us a clearer picture of the underlying mechanisms used in type checking. Jo∃ can also be used to encode existing kinds of existential types in ownership systems with similar benefits.

We would like to use type parameterisation and context quantification to improve the expressivity of multiple ownership and ownership domains systems, and to investigate how existentially quantified contexts can be used in an effects system. It might be useful to extend Jo∃ with subclassing, bounds on type variables, and existential quantification of type variables.

*Acknowledgement* We would like to thank Werner Dietl and the anonymous reviewers for their detailed and useful feedback, and James Noble for ideas and inspiration from discussions on the MOJO project.

## References

1. Marwan Abi-Antoun and Jonathan Aldrich. Ownership Domains in the Real World. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, 2008.
2. Austin Armbruster, Jason Baker, Antonio Cunei, Chapman Flack, David Holmes, Filip Pizlo, Edward Pla, Marek Prochazka, and Jan Vitek. A Real-Time Java Virtual Machine with Applications in Avionics. *Transactions on Embedded Computing Systems*, 7(1):1–49, 2007.
3. Chandrasekhar Boyapati and Martin Rinard. A Parameterized Type System for Race-free Java Programs. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2001.
4. Nicholas Cameron. *Existential Types for Variance — Java Wildcards and Ownership Types*. PhD thesis, Imperial College London, 2009.
5. Nicholas Cameron, Sophia Drossopoulou, and Erik Ernst. A Model for Java with Wildcards. In *European Conference on Object Oriented Programming (ECOOP)*, 2008.

6. Nicholas Cameron, Sophia Drossopoulou, James Noble, and Matthew Smith. Multiple Ownership. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2007.
7. Nicholas Cameron, Erik Ernst, and Sophia Drossopoulou. Towards an Existential Types Model for Java Wildcards. In *Formal Techniques for Java-like Programs (FTfJP)*, 2007.
8. Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.
9. David Clarke. *Object Ownership and Containment.* PhD thesis, School of Computer Science and Engineering, The University of New South Wales, Sydney, Australia, 2001.
10. 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.
11. 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.
12. Werner Dietl, Sophia Drossopoulou, and Peter Müller. Generic Universe Types. In *European Conference on Object Oriented Programming (ECOOP)*, 2007.
13. Werner Dietl and Peter Müller. Universes: Lightweight Ownership for JML. *Journal of Object Technology*, 4(8):5–32, 2005.
14. Giorgio Ghelli and Benjamin Pierce. Bounded existentials and minimal typing. *Theoretical Computer Science*, 193(1-2):75–96, 1998.
15. 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.
16. Atsushi Igarashi and Mirko Viroli. Variant Parametric Types: A Flexible Subtyping Scheme for Generics. *Transactions on Programming Languages and Systems*, 28(5):795–847, 2006.
17. Neel Krishnaswami and Jonathan Aldrich. Permission-Based Ownership: Encapsulating State in Higher-Order Typed Languages. In *Programming Language Design and Implementation (PLDI)*, 2005.
18. Yi Lu and John Potter. On Ownership and Accessibility. In *European Conference on Object Oriented Programming (ECOOP)*, 2006.
19. Yi Lu and John Potter. Protecting Representation with Effect Encapsulation. In *Principles of Programming Languages (POPL)*, 2006.
20. Mads Torgersen and Erik Ernst and Christian Plesner Hansen. Wild FJ. In *Foundations of Object-Oriented Languages (FOOL)*, 2005.
21. John C. Mitchell and Gordon D. Plotkin. Abstract Types have Existential Type. *Transactions on Programming Languages and Systems*, 10(3):470–502, 1988.
22. Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. Generic Ownership for Generic Java. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2006.
23. Tobias Wrigstad. *Ownership-Based Alias Managemant.* PhD thesis, KTH, Sweden, 2006.
24. Tobias Wrigstad and Dave Clarke. Existential Owners for Ownership Types. *Journal of Object Technology*, 6(4), 2007.
25. Tian Zhao, Jens Palsberg, and Jan Vitek. Type-based Confinement. *J. Funct. Program*, 16(1):83–128, 2006.