ABSTRACT

Ownership types structure the heap, and can enforce encapsulation properties which improve security, provide more information for the programmer, and allow for better reasoning about programs. Ownership Generic Java (OGJ) implements ownership types using Java generics and some additional type checking. This allows the programmer to use generics and ownership types in the same programs with little additional syntactic overhead.

We combine Java wildcards (represented formally as existential types) with OGJ to enforce an ownership topology using only features of the Java type system. We demonstrate how the owners-as-dominators encapsulation property can be enforced by a minimal addition to the well-formedness rules for types and classes. We show that the type parameterisation of Java generics — with wildcards — is sufficient to enforce ownership.

1. INTRODUCTION

Ownership types are a mechanism for enforcing an hierarchical structure over the heap. This is done by considering each object to be owned by another object (or to be at the root of the ownership hierarchy). Ownership is statically enforced by the type system, and objects cannot move in the ownership graph at runtime. Ownership systems can support strong encapsulation properties such as owners-as-dominators, which restricts references in the heap according to the ownership tree.

There are several approaches to implementing ownership. One approach [12], is to parameterise types with ownership information, in much the same way as types are parameterised with type information in generic Java. In fact, in OGJ [24], ownership information is type information; the Java type system, with a few additions (mostly to handle the distinguished This owner), is used to define and enforce the ownership structure. OGJ showed that the parametricity of ownership types is the same kind of parametricity as the more familiar and widely studied kind of type parametricity.

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.

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

However, it is philosophically unsatisfactory that OGJ should require machinery in its type system only to handle ownership types. In this paper, we describe and formalise a system that uses only the Java type system to enforce ownership, thus eliminating the need for a special treatment of This. By using wildcards (not dealt with in OGJ) and a simple well-formedness constraint, we can define an ownership hierarchy and enforce the owners-as-dominators encapsulation property. Although we don’t envisage this being a practical way to support ownership, it demonstrates that ownership can be enforced using only standard notions of parametricity — there is nothing exotic about ownership type systems.

The contributions of this paper are: showing how a simple pattern for class declaration and type instantiations can enforce the descriptive aspects of ownership types within the Java 5.0 type system, formalising ownership types within a simple formalisation of Java with wildcards, and formalising the minimal extensions to Java needed to enforce owners-as-dominators. We present syntax, typing rules, and operational semantics; however, as yet we have no soundness proof for our system.

In the next section (Sect. 2) we give some background to this work (Java generics, wildcards, ownership types, and OGJ); we then discuss how Java with wildcards can be used to implement ownership types (Sect. 3); next we formalise this concept in the language WOGJ (Sect. 4); finally we conclude and discuss possible future work (Sect. 5).

2. BACKGROUND

2.1 Generics

In Java, classes, types, and methods may be parameterised with type information. Class and method declarations can take formal type parameters, which can be used in the body of the class or method. Types and method calls1 are annotated with actual type parameters, which instantiate their formal counterparts. For example, a generic list class could be declared as:

```java
class List<X> {
    X get(int i) {...}
    void set(X x) {...}
    List<X> copy() {...}
}
```

1In this paper, we will concentrate on generic classes and types, and ignore generic methods.

OGJ Gone Wild

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

James Noble
Victoria University of Wellington
kjx@ecs.vuw.ac.nz
This class can be instantiated as the type `List<Book>`, which represents a list of books. Calling `get` on an object with this type returns a `Book` object.

### 2.2 Wildcards

Subtyping in Java with generics is invariant. That is, type parameters may not change across subtypes. Assuming the usual hierarchy, `ArrayList<Book>` is a subtype of `List<Book>`; however, `List<Book>` is not a subtype of `List<Object>`. In general it would be unsound to allow such a relationship, but there are certain circumstances where it is safe and convenient to allow some form of variance [16]. Wildcards support such variance in Java [19].

A wildcard is an actual type parameter denoted by ‘?’ A wildcard type is a type parameterised by one or more wildcards. The variance of a wildcard type is given by the wildcards’ bounds. For example, `List<? extends Book>` behaves covariantly: it is a subtype of `List<X extends Book>`, `List<? super Object>` behaves contravariantly: it is a supertype of `List<super Book>`, if that wildcard already has a bound, then the bound used for type checking will be the more precise of the two. For example, given the class declaration `class C<X extends Object>` then in both `C<? extends Book>` and `C<? super Object>`, the wildcard will have the upper bound `Book`.

In order to preserve soundness, the operations which can be performed on a wildcard type must be restricted. Thus, `get` (in the example above) can be called on `List<? extends Book>` and returns a book; however, `set` can only be called with `null` as its parameter. Likewise, `set` on `List<? super Book>` will take a `Book`, but `get` will return an `Object`.

### 2.3 Existential Types

Wildcard types are usually modelled using existential types [25, 8]. `List<?>` is encoded as `∃.List<X>`. Bounds are encoded as bounds on the quantified variables (we write lower and upper bounds inside square brackets `lower upper`) and we use the bottom type, ⊥, and `Object` to represent omitted bounds: `List<? extends Book>` is encoded as `∀X->[⊥ Book].List<X>` and `List<? super Book>` as `∀X->[Book Object].List<X>`.

The subtyping behaviour of wildcard types is given by the usual subtyping properties of existential types, and the access restriction properties are given by packing and unpacking of types (also called closing and opening or existential introduction and elimination).

### 2.4 Ownership Types

Ownership types use mechanisms similar to generic types to manage aliasing relationships between objects. The key here is to keep track of an object’s representation — the inside of an object — and prevent that representation from exposure to the outside [21].

Ownership types [10, 12] do this using two main language constructs. First, every class is parameterised with an `ownership context parameter` (or owner parameter for short) that — for each instance — will record information about that instance’s ownership. We declare a book class `class Book<o>` to indicate that book instances will be owned by “o”. More sophisticated classes can take additional ownership parameters to grant permission to access other objects. A `List<o,i>` class declares two ownership parameters: the owner of the list itself `o`, and the owner of the list items `i`:

```java
class List<o,i> {
  Link<This,i> next;
  Item<i> get (int i) {...}  
  void set (Item<i> x) {...}
  ...
}
```

Note that this code is not type-generic: this list only stores items of class `Item` (or subclasses via subsumption). Note also that this list must have `homogeneous` ownership: all list items must be owned by `i`.

The second language construct needed to implement per-instance ownership is some way to denote and enforce that an object belongs to the instance where it is declared. In this example, the `next` field — part of the list’s internal representation — has ownership type `Link<This,i>`. The special “This” owner (in some systems declared with a `rep` keyword before the type) means that the link objects stored in this field belong to this particular list instance. (The `i` parameter to `Link<This,i>` permits links to store list items with ownership `i`.) The type system will have rules ensuring that types with `This` in their owner position can only be accessed by their owning instance, e.g. by restricting access to calls (implicitly or explicitly) via “This”.

Finally, ownership types require well-formedness constraints that ensure any subsidiary ownership parameters lie outside an object’s main owner — so it is permissible to instantiate e.g. `List<This,other>` — a list that I own but whose items are owned elsewhere, while it is not permissible to instantiate e.g. `List<other,This>` — a list owned by some other object but that holds references to objects I own. Taken together, these restrictions ensure that ownership types support an “owners-as-dominators” discipline, ensuring that an object can only be accessed (directly or indirectly) via its owner: if that access is indirect, it must be only via other objects that are also inside its owner.

Different types of ownership system have been developed to support more flexible topologies [13, 1, 5, 9], or a range of practical applications [11, 28, 3, 4, 6]; implicit ownership systems (also called confined types) use other syntactic techniques or conventions to represent ownership [20, 26, 29]. Some ownership proposals [21, 13, 2] use keywords rather than owner parameters to capture the key instance owner, although theses systems are converging with explicitly parameterised systems [14, 23].

### 2.5 Existential Ownership

Existential types have been combined with ownership types in several ways. They have been used to abstract contexts to increase flexibility [10, 17], support downcasts without requiring ownership information to be retained at runtime [27], and to support subtype variance, both implicitly [13, 9, 18] and explicitly [7]. In these last systems, existential quantification of contexts is used to give variance for ownership types, similar to the use of wildcards in Java. The difference is that variance is with respect to the inside relation, rather than subtyping. For example (in [7]),
as follows:

Potanin et al’s Ownership Generic Java (OGJ) allows the ownership information as a special kind of type information. Some early proposals \[21\] suggest separate parametrisation for owners and types — so a generic list would have three ‘type-ish’ generic parameters: the type of the list items; the ownership of the list items, and the ownership of the list itself, something like `List<X, o, i>`. Unsurprisingly this approach is quite unwieldy in practice — especially as the type and ownership parameters (here the item type `X` and the item ownership `i`) are almost always used together.

To address this problem, Potanin et al \[24, 23\] demonstrated how a single type parameter could carry both ownership and type information simultaneously, by treating ownership information as a special kind of type information. Potanin et al’s Ownership Generic Java (OGJ) allows the definition of a list that is both ownership and type generic as follows:

```java
class List<X, Owner extends World> {
    Link<X, This> next;
    X get (int i) {...}
    void set (X x) {...}
    ...
}
```

First, note that this code sample, although of an OGJ program, is syntactically correct Java code: OGJ is a strict subset of Java, every OGJ program is a Java program. The parameter `X` abstracts both the list items’ type (as in the generic example) and the items’ ownership (as in the ownership type system). All classes also have an additional primary ownership parameter (as in ownership types, but now declared last), but that parameter is now just a generic type parameter (`Owner extends World` is a perfectly valid Java declaration, given an abstract library class `World`). OGJ also requires some basic well-formedness conditions: ownership must be invariant over subclassing/subtyping, and the ownership of additional generic parameters must be outside a type’s primary owner — these conditions can in principle be captured via standard generic type bounds. OGJ thus brings ownership types and conventional generic types much closer together.

The OGJ design subsumes only one of the two language constructs required for ownership — ownership parameterisation is subsumed by generic type parameters. In fact, Potanin et al \[22\] show that generic type parameterisation as found for example in FGJ or Java \[15\] is sufficient to model ownership in static domains, i.e. confined types. OGJ still requires the second language construct, the `This` owner, to enforce per-instance object ownership, and also the supporting special purpose machinery in the type system to give the `This` owner the correct semantics.

In the remainder of this paper, we will argue that an underlying type system with wildcards is strong enough to encode all of OGJ — and thus all object ownership — without any additional, special purpose type system mechanisms.

### 3. OGJ WITH WILDCARDS

Our main contribution is to eliminate the special treatment of the `This` context in OGJ \[24\]. We first handle the purely descriptive function of ownership types, and then in Sect. 3.2 describe how owners-as-dominators can be enforced. We eliminate the special treatment of the `This` context by making it a formal parameter of each class, similar to the `Owner` parameter already present in OGJ. Since `This` is always assumed to be inside `Owner`, it is specified by using `Owner` as the upper bound. Therefore, class declarations have the form

```java
class C<X₀,...,Xₙ, Owner, This extends Owner> {
...
}
```

To ensure the correct behaviour of `This`, it must always be instantiated with an unbounded wildcard, for example: `C<T₀,...,Tₙ, World, ?>>` (where `T₀,...,Tₙ` are types, and instantiations of `C` will be owned by the root context, `World`). Instantiating `This` with a wildcard does not have to be enforced to ensure soundness of the type system or correctness of the ownership hierarchy. However, if `This` can be instantiated by a non-wildcard type, then the behaviour of ownership types is not mimicked correctly: the `This` context is not associated with instances of the declaring class, and therefore contexts do not correspond with objects.

Other than the above innovation, OGJ with wildcards mostly follows OGJ\(^3\); the most important feature of which is the conflation of type and context parameters. For example, we could define a linked list in OGJ with Wildcards as:

```java
class List<X, Owner extends World, This extends Owner> {
    Link<X, This, ?>> next;
    X get (int i) {...}
    void set (X x) {...}
    ...
}
```

where `X` is the type of objects in the list. For example, we can instantiate this list as `List<Book<World, ?>, This, ?>>`, which represents a list of books, where the books are owned by the root context and the list itself is owned by the current context. We can also define lists with unknown owners, for example `List<Book<?>, This, ?>>` (a list of books whose owners are unknown) or `List<Book<World, ?>, This, ?>>` (a list of books whose owners are all `World`, but where the owner of the list is unknown). We can also express uncertainty about the types of objects in the list, for example, `List<?, This, ?>>` (a list of unknown objects). Note that the final “`This`” formal parameter is always instantiated with a plain unbounded wildcard “`>`.”

This can be used as an owner within a class, but cannot be concretely named outside that class, therefore, objects are confined to the part of the ownership hierarchy corresponding to their owner’s context. For example:

\(^3\)We simplify OGJ by not supporting its confinement aspects; we make further simplifications in the formalism, described in the next section.
class C<Owner extends World, This extends Owner> {
    C<This, ?> f1; //field in C's representation
    C<?, ?> f2; //field in some context

    public void m(C<World, ?> x, C<This, ?> y) {
        x.f2 = y.f1;
        //1 OK: owner abstracted
        x.f1 = y.f1;
        //2 error: violates topology
        x.f1 = y;
        //3 error: violates topology
    }
}

In this example, f1 is in the representation of C, indicated by its owner, This. The intention is that ownership types prevent objects in different contexts being mixed, thus disrespecting the ownership hierarchy.

At this stage we are not enforcing an encapsulation property, so we can read x.f1 and store a reference to it by abstracting its owner with a wildcard (expression 1): after capture conversion, x.f1 will have type C<Z, ?>, where Z is fresh; this type is a subtype of C<?, ?>.

In expression 2, x.f1 and y.f1 are instantiations of C, but they are incompatible, because they describe objects which may be in different contexts. The types of x and y will be capture converted to C<World, Z>, where Z is a fresh variable in each case. Therefore x.f1 and y.f1 have types (within the scope of the assignment expression) C<Z1, ?> and C<Z2, ?>, which are incompatible.

Similarly, we cannot assign y (owned by the current context, This) into x.f1 (owned by x), because C<This, ?> is not a subtype of C<Z, ?>, where Z is, again, fresh.

3.1 Some details

Bounds. The formal parameter This must be bounded by Owner to reflect the ownership hierarchy. Owner must be bounded by World to ensure that it represents a context and not a type. When a type is instantiated, the wildcard corresponding to This will inherit the bound from the class declaration, and so will be bounded above by the actual parameter corresponding to Owner. The This wildcard must not be given a lower bound, to ensure that the types do not incorrectly reflect that an object’s context overlaps another context.

Object creation. In Java, new cannot be used to create objects with wildcard type: for example,

    C<World, ? > x = new C<World, ?>()

is illegal. When an object is created, we must use a concrete type to stand for This. The actual Owner parameter can be used for this, in this example,

    C<World, ? > x = new C<World, World>()

This is legal because there is no context between the This and Owner contexts in the ownership hierarchy; therefore any property of the Owner context’s position in the hierarchy also applies to the This context. The new expression must be given the more abstract type with the This parameter hidden by a wildcard immediately. Unfortunately, this must be enforced by the programmer or in the type system, a preprocessor step could allow unsoundness.

3.2 Encapsulation

We can enforce owners-as-dominators in OGJ with wildcards by enforcing a few extra constraints. We require the standard constraint that an object’s owner is inside all of its other actual context parameters; in OGJ with wildcards, this means that the owner must be a subtype of the context parameters. We also require that all wildcards, except those in the This position, are given a lower bound. This prevents an object’s owner being directly or indirectly abstracted which would allow violations of the owners-as-dominators property.

We return to the example given above to illustrate the changes to the system:

```java
class C<Owner extends World, This extends Owner> {
    C<This, ?> f1; //field in C's representation
    C<?, ?> f2; //now illegal

    public void m(C<World, ?> x, C<This, ?> y) {
        x.f2 = y.f1;
        //1 error: violates o-as-d
        x.f1 = y.f1;
        //2 error: violates topology
        x.f1 = y;
        //3 error: violates topology
    }
}
```

Expressions 2 and 3, which caused errors in the descriptive system, still cause errors when enforcing owners-as-dominators. The declaration of field f2 is now illegal because the wildcard in owner position is unbounded. This prevents abstraction of an object’s owner, and so expressions such as expression 1 cannot be written. Note that there is no type error in expression 1, it is the ill-formed type of f2 which causes an error in type checking. Executing expression 1 would violate owners-as-dominators because x, which is in the World context, would hold a reference to an object in y’s representation (y.f1); this reference would allow for a reference chain to y.f1 which is not dominated by y.

The constraints needed to enforce owners-as-dominators cannot be enforced by a pre-processing step, since they must be applied to types at intermediate stages of type checking. However, it would be a fairly simple modification to the type checker to include these extra well-formedness checks.

---

1This could be avoided by enforcing a slightly clunky encoding of object creation which makes use of wildcard capture. For example, to encode new C<World, ?>(), we add the following methods to class C:

```java
private static C<0, X> createAux
    (ArrayList<X> d1, ArrayList<X> d2)
    { return create(new ArrayList<O>(), dummy); }
public static C<0, X> create(ArrayList<X> dummy)
    { return createAux(new ArrayList<X>(), dummy); }
```

We replace the constructor call with C.<World>create(null). The ArrayLists are used only to pass around type parameters, in the call to createAux we must provide both type parameters or neither, since we must infer one, we can prove neither.
4. WOGJ

We formalise the concepts described in the previous section in a simple model for Java with wildcards, WOGJ. This is a simplification of Tame FJ [8], extended with imperative features and the simple well-formedness constraints needed to support ownership types and the owners-as-dominators property.

WOGJ also simplifies Tame FJ [8], it does not support type parameterisation of methods and thus does not support type inference of type parameters at method call sites\textsuperscript{5}. In terms of the formal system, this means that the type rule for method invocation is simpler and that there is no need for the match and sift relations of Tame FJ. Furthermore, WOGJ does not separate the subtyping relation, nor does our rule for well-formed environments include a premise which uses these subtype relations; therefore, WOGJ as presented probably cannot be proved sound. However, soundness can (probably) be proved by restoring these features of Tame FJ. WOGJ is also a simplification with respect to OGJ [24]: we do not support the confinement properties of OGJ, nor placeholder parameters (which are used in our informal description), nor manifest classes.

Ownership types structure the heap, therefore in order to be interesting, WOGJ must support a heap and other imperative features. To this end, we extend Tame FJ with a heap, field assignment, and rules to deal with the semantics of addresses, null, and errors due to null.

Ownership is essentially supported in WOGJ by adding the coding patterns described in the previous section to the well-formedness rules and syntax. The rule for well-typed classes is modified so that these changes can be assumed when type checking the body of a class.

4.1 Syntax

The syntax of WOGJ is given in Fig. 1, runtime entities are surrounded by a grey box. The syntax for expressions and types is mostly standard and follows Tame FJ and FJ-like calculi with assignment, e.g., OGJ [24] or Joη [7]. As with these systems, we will elide empty quantifying environments and type parameter lists, e.g., we use ι as a shorthand for ∃ι.R-Class. Class types are extended with the distinguished class World, all types that represent contexts in the ownership hierarchy must inherit from World. Types are also extended with TThis, which is only used in the operational semantics to associate a type parameter with the TThis context. No corresponding change to the Java runtime system would be required since type parameters are erased [15]. Class types are forced to take at least two actual type parameters to represent the Owner and TThis contexts; the corresponding formal parameters have distinguished names and, in the case of TThis, a syntactically fixed bound. Following OGJ, we separate type variables which represent formal contexts (0) from those which represent types (X). This is a convenience (only used in T-Class in Fig. 8) and this distinction is not essential; the two can be distinguished without this distinction since the bound of an 0 extends World whereas the bound of an X extends Object. Where it is clear from the context, we will assume that the syntactic category X includes 0.

In the following figures, we use a grey box to highlight parts of rules which are used to support ownership in WOGJ.

5Note that we do support capture conversion of receivers.

4.2 Operational Semantics

The interesting operational semantics rules are given in Fig. 2; rules for congruence, null, and error propagation are in the appendix. The rules for field access and assignment, and method invocation are standard: addresses are looked up in the heap and a result computed by looking up the value in the field or method body, and in the case of assignment, updating the heap.

R-New creates a new object record in the heap at a fresh address, all fields are initialised to null and can be set by field assignment. The interesting detail is that the type in the heap is not identical to the type being instantiated: the special type TThis, is used in the heap; it is unique for each address. We do this to ensure soundness by ensuring an object's TThis context is always abstracted, whilst avoiding instantiating wildcard types, which is illegal in Java. This type represents the representation of an object. As we will see shortly, a similar transformation takes place in the type rule for object creation.

4.3 Auxiliary Functions

The auxiliary functions in WOGJ (given in Fig. 3) are used to find the owner for a given type or address. An owner is represented as a type which is a subtype of World. To find the owner of an object in the heap (owner\textsubscript{T}(l)), the object's address is looked up in the heap. The object record in the heap includes the object's type (which will be a class type) and the actual type parameter in the owner position is returned.

The owner of a type (owner\textsubscript{T ∆}(T)) is found for class types by taking the actual type parameter in the owner position. The operation is complicated by existential quantification: if the type parameter is quantified then we must find (using glb) a transitive lower bound of this type parameter which is not quantified. The definition of owner follows Joη [7] and ensures that owner is downwardly conservative, i.e., owner may give a context which is inside the precise owner, but never one which is outside.

4.4 Subtyping

Subtyping is defined in Fig. 4 and follows Tame FJ [8] (without the division into subclassing etc.). Most rules are standard; the only interesting rule is S-Env, which gives subtyping between existential types (which model wildcard types). The intuition is that the subtype will have more precise type parameters than the supertype: either an unquantified variable within the bounds of the corresponding quantified variable on the right hand side, or a quantified variable with more restrictive bounds. For example, ∃ι.R and ∃X.[.R .].C<X> are subtypes of ∃X.[.R .].C<X>.

4.5 Well-formedness

Well-formedness of types and environments is defined in Fig. 5, again, these rules mostly follow Tame FJ. One structural difference is that we merge the rules for well-formed existential types and class types. This is done so that the rule can be aware which type parameters are existentially quantified. The added premises in the rules for well-formed types ensure that the type parameter in the TThis position is quantified and has no lower bound, that all other parameters are either unquantified or bounded below, and that any type parameters which represent contexts are outside the owner context.
\[ e ::= \text{null} \mid x \mid e.f \mid e.f = e \mid e.m(\overline{x}) \mid \text{new } N \quad \text{expressions} \]

\[ Q ::= \text{class } C<\Omega, T, \text{Owner}, \text{This} < V, \text{H} > \mid \text{Object}<\text{Owner}, T_{\text{This},}> \mid \text{World}< \quad \text{class declarations} \]

\[ M ::= <X,T> t(m(e)) \quad \text{method declarations} \]

\[ N ::= C<T, X, \text{Owner}, T, \text{This}, T> \mid \text{null} \mid \text{err} \quad \text{values} \]

\[ R ::= N \mid X \quad \text{non-existential types} \]

\[ T, U ::= \exists \Delta . N \mid \exists \emptyset . X \mid \emptyset . \text{This}, \emptyset . \text{Owner} \quad \text{types} \]

\[ \Delta ::= X \rightarrow [B, b_n] \quad \text{type environments} \]

\[ \gamma ::= x \mid \overline{x} \mid \text{null} \quad \text{vars and addresses} \]

\[ \Gamma ::= \gamma : T \quad \text{var environments} \]

\[ H ::= \overline{i} \rightarrow \{N; f \rightarrow v\} \quad \text{heaps} \]

\[ x \quad \text{variables} \]

\[ C \quad \text{classes} \]

\[ X, Y \quad \text{type variables} \]

\[ O, \text{Owner}, \text{This} \quad \text{type variables (owners)} \]

\[ i \quad \text{addresses} \]

---

**Figure 1:** Syntax of WOGJ; runtime entities are in grey.

**Figure 2:** WOGJ reduction rules.

**Auxiliary Functions:**

\[ \text{owner}_H(i) = T \quad \text{owner}_\Delta(T) = T \]

\[ \Delta \vdash X < V \quad \text{owner}_\Delta(X) = X \]

\[ \Delta \vdash X < V \quad \text{owner}_\Delta(X) = \bot \]

\[ \text{owner}_{\exists \Delta'.C<T, \text{Owner}, T_{\text{This},}>}(i) = i \]

---

**Figure 3:** Auxiliary functions for WOGJ.
class C<X<T>, Y, Z> < N {...}

\[ \Delta \vdash \exists \Delta', \langle \text{C} \rangle T < : \exists \Delta'. (T/X) N \]

(S-REFL)

\[ \Delta \vdash \bot : B \]

(S-BOTTOM)

\[ \Delta \vdash B \ll B' \]

(S-TRANS)

\[ \delta(\Delta') \cap f(v(\exists X : [B; B_u]. N)) = \emptyset \quad f(v(T)) \subseteq \delta(\Delta', \Delta') \]

(f-extend)

\[ \Delta, \Delta' \vdash [T/X] B \ll T \]

(f-bnd)

\[ \Delta, \Delta' \vdash T : \exists \Delta'. (T/X) B \]

(f-extend)

\[ \Delta \vdash B < : B' \]

(S-BOUN)

\[ \Delta \vdash \exists \Delta', \langle \text{C} \rangle T < : \exists \Delta'. (T/X) N \]

(XS-ENV)

\[ \Delta \vdash \bot < : B \]

(S-BOUN)

\[ \Delta \vdash B < : \exists \Delta'. X \]

(S-BOUN)

---

**Figure 4:** WOGJ subtyping.

<table>
<thead>
<tr>
<th>Well-formed types: ( \Delta \vdash B \text{ OK, } \Delta \vdash P \text{ OK, } \Delta \vdash R \text{ OK} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Delta \vdash X \text{ OK} ) \hspace{1cm} ( \Delta \vdash \bot \text{ OK} ) \hspace{1cm} ( \Delta \vdash \exists \text{World}\text{&lt;}	ext{&lt;} \text{OK} )</td>
</tr>
<tr>
<td>(F-VAR) \hspace{2cm} (F-BOTTOM) \hspace{3cm} (F-WORLD)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Well-formed type environments: ( \Delta \vdash \Delta \text{ OK} )</th>
</tr>
</thead>
</table>
| \( \Delta \vdash \exists \Delta', \langle \text{C} \rangle T < : \exists \Delta'. N \) \hspace{1cm} \( \Delta, \Delta' \vdash T : \exists \Delta'. (T/X) N \)
| (F-Extend-Class) |

\( T_{\text{Obj}} \in \delta(\Delta) \Rightarrow \delta'(T_{\text{Obj}}) = [T; B] \)

\( \delta'(T_{\text{This}}) = [\bot; B'] \)

| \( \forall T_e \in T_0,...,T_{n-1} : \Delta, \Delta' \vdash T_e : [T/X] T_a \) |

\( \forall T_e \in T_0,...,T_{n-2} : \Delta, \Delta' \vdash T_{n-2} < : \text{owner}\Delta, \Delta(T_i) \)

| \( \Delta \vdash X : [B; B_u], \Delta', \Delta' \vdash B \text{ OK} \) \hspace{1cm} \( \Delta, X : [B; B_u], \Delta' \vdash B \text{ OK} \) |
| (F-ENV-EMPTY) |

| \( \Delta \vdash B < : B' \) \hspace{1cm} \( \Delta \vdash B < : B_u \) \hspace{1cm} \( \Delta, X : [B; B_u], \Delta' \vdash B \text{ OK} \) |

---

**Figure 5:** WOGJ well-formed types and type environments.

Well-formed environments ensure that all bounds are well-formed and that the lower bound is a subtype of the upper bound."

### 4.6 Type Checking

Type checking is defined by the rules in Fig. 6; they are mostly standard. Following Tame FJ, we use bounding environments to ensure that any unpacked type variables are re-packed or eliminated by subtyping. We elide a full description here, since the type rules are almost identical to Tame FJ. In Fig. 7 we give an example of field access (\( x, \text{datum} \) where \( x \) is an instance of \( \text{List}<\text{?}<\text{Book}<\text{World}>, \text{World}, \text{?} > \text{<} > \) ) to give an idea of their operation.

The rule for object creation is a little unusual: the expression is not assigned the type which is actually written. For example, \( \text{new C}<\text{A}, \text{B}, \text{C}> \) has type \( \exists X : [\bot; B]. C<\text{A}, \text{B}, \text{x}> \). This ensures that all types in the system are well-formed (which requires that the \( \text{This} \) context is instantiated with.

"Unfortunately this sanity check is not strong enough to prevent spurious assumptions being used to derive unsound subtypes. For this we require a stronger constraint which is defined using the subclassing relation of Tame FJ [8].

"This use of bounds means that they must be put at the end of the type environment to prevent a forward reference.
Expression typing: $\Delta; \Gamma \vdash e : T \mid \Delta$

Method typing: $\Delta \vdash M \text{ in } C$

Class typing: $\Delta \vdash Q \text{ OK}$

Well-formed heaps and configurations: $\Delta, H \vdash \text{ok} \implies H \in \mathcal{H}, \Delta, H \vdash e \text{ ok}$

Figure 6: WOGJ expression typing rules.

Figure 7: Example derivation.

Figure 8: WOGJ class and method typing rules.

Figure 9: WOGJ well-formed heaps and configurations.
outside the actual parameter in $\text{Owner}$ position.

### 4.7 The Heap

Well-formed heaps and configurations are defined in Fig. 9. F-HEAP ensures that all types in the heap are well-formed, all references refer to addresses which are in the heap and which have types corresponding with the references, and that the heap satisfies owners-as-dominators.

Owners-as-dominators is specified by ensuring that the unique type $\text{This}$ is a subtype of the owner of all values referenced by $\iota$; which ensures that $\iota$ is inside the owner of these references within the ownership hierarchy.

Methods and field lookup functions, and rules for using the heap as a variable environment are mostly standard and have been relegated to the appendix. The only interesting point is that, when using the heap as an environment, for each location, $\iota$, in the heap, we add the assumption $\text{This} \leftarrow (\bot \cdot \text{Towner})$ to the type environment, where $\text{Towner}$ is the actual type parameter in the owner position of $\iota$’s type. Therefore, we can always assume the expected relation between an object and its owner. Adding this assumption to the environment is only required to prove that WOGJ satisfies owners-as-dominators, and, due to type erasure, would not need to be reflected at runtime in an implementation.

### 4.8 Properties

The relevant properties for WOGJ are type safety (to be shown in the usual way by progress and preservation theorems) and owners-as-dominators (proved as part of preservation). We suspect that these will both hold with some insubstantial changes to the calculus, but cannot be sure as we have not yet attempted proofs. We believe the proof for type soundness will follow those of Tame FJ [8] and Jo$\exists$ \cite{Joexists}, and for owners-as-dominators of Jo$\exists$ and OGJ \cite{jogj}.

#### Conjecture — Progress

For any $\Delta$, $\mathcal{H}$, $e$, $\tau$, if $\emptyset; \mathcal{H} \vdash e : \tau \mid \Delta$ then either there exists $e'$ and $\mathcal{H}'$ where $\mathcal{H}; e \sim \mathcal{H}'; e'$ or there exists $\nu$ where $e = \nu$.

#### Conjecture — Preservation

For any $\mathcal{H}, \mathcal{H}'$, $e$, $e'$, $\tau$, if $\emptyset; \mathcal{H} \vdash e : \tau \mid \emptyset$ and $\mathcal{H}; e \sim \mathcal{H}'; e'$ then $\emptyset; \mathcal{H} \vdash e' : \tau \mid \emptyset$.

#### Conjecture — Owners-as-dominators

For any $\Delta$, $\mathcal{H}$, if $\Delta \vdash \mathcal{H}$ then $\forall \iota : \{ \nu; [\mathcal{H} \sim \nu] \} \in \mathcal{H}, \forall \nu, \in \tau : \nu \neq \text{null} \Rightarrow \Delta; \mathcal{H} \vdash \text{This}, \prec; \owner_{\mathcal{H}}(\nu)$.

Most ownership languages, including OGJ, forbid setting fields in an object’s representation to null. This is a sensible feature of encapsulation, but is not strictly required to satisfy owner-as-dominators. This property is not supported in WOGJ, since it does not emerge naturally from the use of wildcards, but could be enforced by restricting the types which null can assume in T-NULL.

### 5. CONCLUSION AND FUTURE WORK

In this paper we have shown how generics, wildcards, and a simple, syntactic well-formedness constraint can be used to implement ownership. With some very small additions to the rules for well-formed types (which does not affect the typing rules, and which uses type information provided strictly by the Java type system) we can enforce the owners-as-dominators encapsulation property. We have demonstrated that ownership types and their required behaviour can be represented by Java types.

**Future Work.** We are working on a full formalisation in the style of Tame FJ, and a soundness proof for this system. We are also investigating adding wildcards to the implementation of OGJ. There seems little benefit in implementing the extra checking that would enable ownership to be enforced in Java, since native ownership — perhaps in the style of OGJ — is much easier to use. In the longer term, we are interested in how the observations of this work could improve foundational models for ownership systems.

**Acknowledgements.** We would like to thank Alex Potanin for helping us to better understand the intricacies of the OGJ type system, and the anonymous reviewers for their useful comments. The first author’s work was funded by a Build IT Postdoctoral fellowship.

### 6. REFERENCES


APPENDIX

A. ELIDED RULES
Error propagation:

Null pointer exceptions:

Congruence rules:

null.f; H \sim \text{err}; H
(R-Field-Null)

null.f = e; H \sim \text{err}; H
(R-Assign-Null)

null.m(\varnothing); H \sim \text{err}; H
(R-Invoke-Null)

null.f; H \sim \text{err}; H
(R-Field-Err)

e.f = e; H \sim \text{err}; H
(R-Assign-Err)

null.m(\varnothing); H \sim \text{err}; H
(R-Invoke-Err)

null.f; H \sim \text{err}; H
(R-Field-Err)

e.f = e; H \sim \text{err}; H
(R-Assign-Err)

null.m(\varnothing); H \sim \text{err}; H
(R-Invoke-Err)

Figure 10: WOGJ reduction rules for congruence, null, and error propagation.

Lookup Functions

fields(Object) = \emptyset

class C\langle T_0, \ldots \rangle < D \ldots > \{ U T; R \}

fields(\emptyset) = \emptyset, t

class C\langle T_0, \ldots \rangle < N \{ U T; R \} f \notin T

fields(\emptyset) = \emptyset, f

class C\langle T_0, \ldots \rangle < N \{ U T; R \} m \notin M

mBody(m, C\langle T \rangle) = mBody(m, \{ T/X \} R)

class C\langle T_0, \ldots \rangle < N \{ U T; R \} m \notin M

mBody(m, C\langle T \rangle) = mBody(m, \{ T/X \} R)

class C\langle T_0, \ldots \rangle < N \{ U T; R \} m \notin M

mType(m, C\langle T \rangle) = mType(m, \{ T/X \} R)

Figure 11: Method and field lookup functions for WOGJ.

\begin{align*}
H = \iota \rightarrow \{ C: T, T\text{Owner}, T\text{This}, \ldots \} \\
\Delta, \text{This}, \rightarrow \{ T\text{Owner}, T\text{This}, \ldots \} \\
(H-S)
\end{align*}

Figure 12: Using the heap as an environment in WOGJ.
We don't explicitly state everything that is and is not allowed, so please don't try to skirt the rules. Thank you! For dating, 1-on-1, soulmates, friends, and hook-ups please go to /r/r4r or /r/dirtyr4r. GW Hot All Time | GW Hot This Week. If you host your image at imgur.com, be sure to create an account there so that you have control over your assets. Girls gone Wild. 14,746 likes · 26 talking about this. Prodotto dalla V.L.M.production, uno spettacolo completamente ballato e cantato dal vivo con brani... Scelto da Harley Davidson, da marchi di birra come Heineken, Forst ed HB, da Street Food in tutta It