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

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 addtional 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.

IWACO '09, July 6 2009, Genova, Italy

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 calls[1] are annotated with actual type parameters, which instantiate their formal counterparts. For example, a generic list class could be declared as:

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

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

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<? extends Object>`; `List<? super Object>` behaves contravariantly: it is a subtype of `List<? super Book>`; `List<?>` is bivariant: it is a supertype of any list.

If a formal parameter has a bound, then this bound is inherited by the corresponding wildcard. 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 Book>`, then in both `C<?>` and `C<? extends 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 list 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 $\exists$`X.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, $\bot$, and `Object` to represent omitted bounds): `List<? extends Book>` is encoded as $\exists$`X`$\rightarrow$`[`$\bot$ `Book].List<X>` and `List<? super Book>` as $\exists$`X`$\rightarrow$`[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`:

```
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]),

$\exists o \rightarrow$[⊥ o1].Book<o>[2] is a subtype of $\exists o \rightarrow$[⊥ o2].Book<o>, if o1 is inside o2.

2.6 OGJ

Classical ownership types embody the irony that while the ownership is generic, the underlying types are not: the code example in section 2.4 is ownership-generic (lists and items may have different owners in different instances) but not type generic (the list is always a list of Items). 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:

```
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.

[2]Here ⊥ represents the bottom context, not the bottom type.

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

$$\texttt{class C<X}_0\texttt{,...,X}_n\texttt{, Owner, This extends Owner>}$$

. To ensure the correct behaviour of This, it must always be instantiated with an unbounded wildcard, for example: $\texttt{C<T}_0\texttt{,...,T}_n\texttt{, World, ?>}$ (where $\texttt{T}_0\texttt{,...,T}_n$ 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:

```
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, ?>, ?, ?> (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) $\texttt{C<Z}_1\texttt{, ?>}$ and $\texttt{C<Z}_2\texttt{, ?>}$, which are incompatible.

Similarly, we can not 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[4].

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:

```
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.

[4]This 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`:

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

We replace the constructor call with `C.<World>create(null)`. The `ArrayList`s 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 provie 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[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 Jo∃ [7]. As with these systems, we will elide empty quantifying environments and type parameter lists, e.g., we use `C` as a shorthand for ∃∅.`C<`∅`>`. 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 $\texttt{This}_\iota$, which is only used in the operational semantics to associate a type parameter with the `This` 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 `This` contexts; the corresponding formal parameters have distinguished names and, in the case of `This`, a syntactically fixed bound. Following OGJ, we separate type variables which represent formal contexts (`O`) from those which represent types (`X`). This is merely 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 `O` 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 `O`.

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

[5]Note 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 calculated 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 $\texttt{This}_\iota$ is used in the heap; it is unique for each address. We do this to ensure soundness by ensuring an object's `This` 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 ($owner_{\mathcal{H}}(\iota)$), 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 ($owner_\Delta(\texttt{T})$) 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 Jo∃ [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, `C<A>` and ∃`X`→`[`⊥ `A].C<X>` are subtypes of ∃`X`→`[`⊥ `Object].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 `This` 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	::=	null | x | e.f | e.f = e | e.m($\overline{\text{e}}$) | new N	*expressions*
Q	::=	class C<$\overline{\text{O}}\lhd$ T,$\overline{\text{X}\lhd \text{T}}$, Owner $\lhd$ T, This $\lhd$ $\exists\emptyset$.Owner> $\lhd$ N {$\overline{\text{T f;}}$ $\overline{\text{M}}$}	*class declarations*
M	::=	<$\overline{\text{X}\lhd \text{T}}$> T m($\overline{\text{T x}}$) {return e;}	*method declarations*
v	::=	ι | null | err	*values*
N	::=	C<$\overline{\text{T}}$, T_{Owner}, T_{This}> | Object<T_{Owner}, T_{This}> | World<>	*class types*
R	::=	N | X	*non-existential types*
T, U	::=	$\exists\Delta$.N | $\exists\emptyset$.X | $\exists\emptyset$.This_ι	*types*
Δ	::=	$\overline{\text{X}\rightarrow[\text{B}_l\ \text{B}_u]}$	*type environments*
γ	::=	x | ι | null	*vars and addresses*
Γ	::=	$\overline{\gamma:\text{T}}$	*var environments*
$\mathcal{H}$	::=	$\overline{\iota\rightarrow\{\text{N; }\overline{\text{f}\rightarrow\text{v}}\}}$	*heaps*
Γ	::=	$\overline{\text{x:T}}$	*variable environments*
B	::=	T | $\bot$	*bounds*
x			*variables*
C			*classes*
X, Y			*type variables*
O, Owner, This			*type variables (owners)*
ι			*addresses*

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

Computation rules: $\boxed{\text{e};\mathcal{H}\rightsquigarrow\text{e};\mathcal{H}}$

$$\frac{\mathcal{H}(\iota)=\{\text{R};\overline{\text{f}\rightarrow\text{v}}\}}{\iota.\text{f}_i;\mathcal{H}\rightsquigarrow\text{v}_i;\mathcal{H}}\ (\text{R-Field})$$

$$\frac{\mathcal{H}(\iota)=\{\text{R};\overline{\text{f}\rightarrow\text{v}}\}\qquad\mathcal{H}'=\mathcal{H}[\iota\mapsto\{\text{R};\overline{\text{f}\rightarrow\text{v}}[\text{f}_i\mapsto\text{v}]\}]}{\iota.\text{f}_i\ =\ \text{v};\mathcal{H}\rightsquigarrow\text{v};\mathcal{H}'}\ (\text{R-Assign})$$

$$\frac{\mathcal{H}(\iota)\ undefined\qquad fields(\text{C})=\overline{\text{f}}\qquad\mathcal{H}'=\mathcal{H},\iota\rightarrow\{\text{C<}\overline{\text{T}}\text{, }\text{This}_\iota\text{>};\overline{\text{f}\rightarrow\text{null}}\}}{\text{new C<}\overline{\text{T}}\text{, }\text{T}_{This}\text{>};\mathcal{H}\rightsquigarrow\iota;\mathcal{H}'}\ (\text{R-New})$$

$$\frac{\mathcal{H}(\iota)=\{\text{R};...\}\qquad mBody(\text{m},\text{R})=(\overline{\text{x}};\text{e})}{\iota.\text{m}(\overline{\text{v}});\mathcal{H}\rightsquigarrow[\overline{\text{v/x}},\ \iota/\text{this}]\text{e};\mathcal{H}}\ (\text{R-Invk})$$

Figure 2: WOGJ reduction rules.

Auxiliary Functions: $\boxed{owner_{\mathcal{H}}(\iota)=\text{T}}$ $\boxed{owner_{\Delta}(\text{T})=\text{T}}$

$$\frac{\mathcal{H}(\iota)=\{\text{C<}\overline{\text{T}}\text{, }\text{T}_{Owner}\text{, }\text{This}_\iota\text{>}\ ...\}}{owner_{\mathcal{H}}(\iota)=\text{T}_{Owner}}$$

$$\frac{\text{B}\notin dom(\Delta)}{glb_{\Delta}(\text{B})=\text{B}}\qquad\frac{\Delta(\text{X})=[\text{B}_l\ \text{B}_u]}{glb_{\Delta}(\text{X})=glb_{\Delta}(\text{B}_l)}$$

$$\frac{\Delta\vdash\text{X}<:\exists\emptyset.\text{World<>}}{owner_{\Delta}(\text{X})=\text{X}}\qquad\frac{\Delta\vdash\text{X}<:\exists\Delta'.\text{Object<T}_{Owner}\text{, T}_{This}\text{>}}{owner_{\Delta}(\text{X})=glb_{\Delta'}(\text{T}_{Owner})}$$

$$\frac{\Delta(\text{This}_\iota)=[\bot\ \text{T}]}{owner_{\Delta}(\text{This}_\iota)=\text{T}}\qquad\frac{}{owner_{\Delta}(\exists\Delta'.\text{C<}\overline{\text{T}}\text{, T}_{Owner}\text{, T}_{This}\text{>})=\exists\Delta'.glb_{\Delta'}(\text{T}_{Owner})}$$

$$\frac{}{owner_{\Delta}(\exists\emptyset.\text{World<>})=\exists\emptyset.\text{World<>}}$$

Figure 3: Auxiliary functions for WOGJ.

$$\frac{\texttt{class C<}\overline{\texttt{X}\lhd\ \texttt{T}_u}\texttt{>}\ \lhd\ \texttt{N}\ \{\ldots\}}{\Delta \vdash \exists\Delta'.\texttt{C<}\overline{\texttt{T}}\texttt{>} <: \exists\Delta'.\texttt{[}\overline{\texttt{T/X}}\texttt{]N}}\ \text{(S-Sub-Class)} \qquad \frac{}{\Delta \vdash \bot <: \texttt{B}}\ \text{(S-Bottom)} \qquad \frac{}{\Delta \vdash \texttt{B} <: \texttt{B}}\ \text{(S-Reflex)}$$

$$\frac{\Delta \vdash \texttt{B} <: \texttt{B}'' \quad \Delta \vdash \texttt{B}'' <: \texttt{B}'}{\Delta \vdash \texttt{B} <: \texttt{B}'}\ \text{(S-Trans)} \qquad \frac{dom(\Delta') \cap fv(\exists\overline{\texttt{X}\rightarrow\texttt{[B}_l\ \texttt{B}_u\texttt{]}}.\texttt{N}) = \emptyset \qquad fv(\overline{\texttt{T}}) \subseteq dom(\Delta, \Delta') \quad \Delta, \Delta' \vdash \overline{\texttt{[T/X]B}_l <: \texttt{T}} \qquad \Delta, \Delta' \vdash \overline{\texttt{T} <: \texttt{[T/X]B}_u}}{\Delta \vdash \exists\Delta'.\texttt{[}\overline{\texttt{T/X}}\texttt{]N} <: \exists\overline{\texttt{X}\rightarrow\texttt{[B}_l\ \texttt{B}_u\texttt{]}}.\texttt{N}}\ \text{(XS-Env)} \qquad \frac{\Delta(\texttt{X}) = \texttt{[B}_l\ \texttt{B}_u\texttt{]}}{\Delta \vdash \exists\emptyset.\texttt{X} <: \texttt{B}_u \quad \Delta \vdash \texttt{B}_l <: \exists\emptyset.\texttt{X}}\ \text{(S-Bound)}$$

Figure 4: WOGJ subtyping.

Well-formed types: $\boxed{\Delta \vdash \texttt{B}\ \text{OK}, \Delta \vdash \texttt{P}\ \text{OK}, \Delta \vdash \texttt{R}\ \text{OK}}$

$$\frac{\texttt{X} \in \Delta}{\Delta \vdash \texttt{X}\ \text{OK}}\ \text{(F-Var)} \qquad \frac{}{\Delta \vdash \bot\ \text{OK}}\ \text{(F-Bottom)} \qquad \frac{}{\Delta \vdash \exists\emptyset.\texttt{World<>}\ \text{OK}}\ \text{(F-World)} \qquad \frac{\texttt{T}_{Owner} \in dom(\Delta') \Rightarrow \Delta'(\texttt{T}_{Owner}) = \texttt{[T B]} \quad \Delta'(\texttt{T}_{This}) = \texttt{[}\bot\ \texttt{B}'\texttt{]}}{\Delta \vdash \exists\Delta'.\texttt{Object<T}_{owner}\texttt{, T}_{This}\texttt{>}\ \text{OK}}\ \text{(F-Object)}$$

$$\frac{\Delta \vdash \Delta'\ \text{OK} \quad \texttt{class C<}\overline{\texttt{X}\lhd\ \texttt{T}_u}\texttt{>}\ \lhd\ \texttt{N}\ \{\ldots\} \quad \Delta, \Delta' \vdash \overline{\texttt{T}}\ \text{OK} \quad \Delta, \Delta' \vdash \overline{\texttt{T} <: \texttt{[}\overline{\texttt{T/X}}\texttt{]T}_u} \quad |\overline{\texttt{T}}| = n \quad \texttt{T}_{n-2} \in dom(\Delta') \Rightarrow \Delta'(\texttt{T}_{n-2}) = \texttt{[T B]} \quad \texttt{T}_{n-1} = \exists\emptyset.\texttt{This}_\iota \vee \Delta'(\texttt{T}_{n-1}) = \texttt{[}\bot\ \texttt{B}'\texttt{]} \quad \forall \texttt{T}_i \in \texttt{T}_0 \ldots \texttt{T}_{n-3} : \Delta, \Delta' \vdash \texttt{T}_{n-2} <: owner_{\Delta,\Delta'}(\texttt{T}_i)}{\Delta \vdash \exists\Delta'.\texttt{C<}\overline{\texttt{T}}\texttt{>}\ \text{OK}}\ \text{(F-Exist-Class)}$$

Well-formed type environments: $\boxed{\Delta \vdash \Delta\ \text{OK}}$

$$\frac{}{\Delta \vdash \emptyset\ \text{OK}}\ \text{(F-Env-Empty)} \qquad \frac{\Delta, \texttt{X}\rightarrow\texttt{[B}_l\ \texttt{B}_u\texttt{]}, \Delta' \vdash \texttt{B}_l\ \text{OK} \quad \Delta, \texttt{X}\rightarrow\texttt{[B}_l\ \texttt{B}_u\texttt{]}, \Delta' \vdash \texttt{B}_u\ \text{OK} \quad \Delta \vdash \texttt{B}_l <: \texttt{B}_u \quad \Delta, \texttt{X}\rightarrow\texttt{[B}_l\ \texttt{B}_u\texttt{]} \vdash \Delta'\ \text{OK}}{\Delta \vdash \texttt{X}\rightarrow\texttt{[B}_l\ \texttt{B}_u\texttt{]}, \Delta'\ \text{OK}}\ \text{(F-Env)}$$

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[6].

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 repacked 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.datum` where `x` is an instance of `List<?⊲ Book<World>, World, ?>`[7]) 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, `new C<A, B, C>` has type $\exists\texttt{X}\rightarrow\texttt{[}\bot\ \texttt{B]}.\texttt{C<A, B, X>}$. This ensures that all types in the system are well-formed (which requires that the `This` context is instantiated with a wildcard, represented as an existentially quantified variable). In ownership terms, the rule ensures that the owner of values in an object's representation can never be named outside of that object.

Our treatment of object creation is sound because the given type parameter ($\texttt{T}_{This}$) is never used, and the type rule corresponds to the operational semantics for object creation described in Sect. 4.2. Because the $\texttt{This}_\iota$ parameter used in the object record in the heap is always considered to be inside the current object's actual owner (`B` in the above example, see Fig. 9), the synthesised type is always a supertype of the object's type in the heap (by S-Env).

Type rules for methods and classes are given in Fig. 8. Again, these are mostly standard; the only differences here are in T-Class. The added premises ensure invariance of `Owner` and `This` with respect to inheritance, and that the bound on the owner parameter represents a context. To create a type environment to type check the body of the class, type parameters ($\overline{\texttt{X}}$), `Owner`, and `This` are treated as standard parameters, and type parameters representing contexts ($\overline{\texttt{O}}$) are given the lower bound `Owner`[8]. These lower bounds reflect F-Class, which enforces that all actual contexts are

[6]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].

[7]Note that this type isn't well-formed because the first wildcard does not have a lower bound, however, this is only a concern if we wish to enforce owners-as-dominators.

[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: $\boxed{\Delta;\Gamma \vdash \mathtt{e} : \mathtt{T} \mid \Delta}$

$$\frac{\Delta \vdash \mathtt{T}\ \text{OK}}{\Delta;\Gamma \vdash \mathtt{null} : \mathtt{T} \mid \emptyset}\ \text{(T-Null)} \qquad \frac{}{\Delta;\Gamma \vdash \gamma : \Gamma(\gamma) \mid \emptyset}\ \text{(T-Var)}$$

$$\frac{\mathtt{T}_{new} = \exists \mathtt{X} \rightarrow [\bot\ \mathtt{T}_{Owner}].\mathtt{C}\langle \overline{\mathtt{T}},\ \mathtt{T}_{Owner},\ \mathtt{X}\rangle \quad \Delta \vdash \mathtt{T}_{new}\ \text{OK}}{\Delta;\Gamma \vdash \mathtt{new}\ \mathtt{C}\langle \overline{\mathtt{T}},\ \mathtt{T}_{Owner},\ \mathtt{T}_{This}\rangle : \mathtt{T}_{new} \mid \emptyset}\ \text{(T-New)}$$

$$\frac{\Delta;\Gamma \vdash \mathtt{e} : \exists \Delta'.\mathtt{N} \mid \emptyset \quad fType(\mathtt{f},\mathtt{N}) = \mathtt{T}}{\Delta;\Gamma \vdash \mathtt{e.f} : \mathtt{T} \mid \Delta'}\ \text{(T-Field)}$$

$$\frac{\Delta;\Gamma \vdash \mathtt{e} : \exists \Delta'.\mathtt{N} \mid \emptyset \quad fType(\mathtt{f},\mathtt{N}) = \mathtt{T} \quad \Delta;\Gamma \vdash \mathtt{e'} : \mathtt{T'} \mid \emptyset \quad \Delta,\Delta' \vdash \mathtt{T'} <: \mathtt{T}}{\Delta;\Gamma \vdash \mathtt{e.f = e'} : \mathtt{T'} \mid \Delta'}\ \text{(T-Assign)}$$

$$\frac{\Delta;\Gamma \vdash \mathtt{e} : \exists \Delta'.\mathtt{N} \mid \emptyset \quad \Delta;\Gamma \vdash \overline{\mathtt{e} : \mathtt{U'} \mid \emptyset} \quad mType(\mathtt{m},\mathtt{N}) = \overline{\mathtt{U}} \rightarrow \mathtt{U} \quad \Delta,\Delta' \vdash \overline{\mathtt{U'} <: \mathtt{U}}}{\Delta;\Gamma \vdash \mathtt{e.m}(\overline{\mathtt{e}}) : [\overline{\mathtt{T}/\mathtt{Y}}]\mathtt{U} \mid \Delta'}\ \text{(T-Invk)}$$

$$\frac{\Delta;\Gamma \vdash \mathtt{e} : \mathtt{U} \mid \Delta' \quad \Delta,\Delta' \vdash \mathtt{U} <: \mathtt{T} \quad \Delta \vdash \Delta'\ \text{OK} \quad \Delta \vdash \mathtt{T}\ \text{OK}}{\Delta;\Gamma \vdash \mathtt{e} : \mathtt{T} \mid \emptyset}\ \text{(T-Subs)}$$

Figure 6: WOGJ expression typing rules.

$$\frac{\dfrac{\emptyset;\Gamma \vdash \mathtt{x} : \exists \mathtt{X} \rightarrow [\bot\ \mathtt{Book\langle World\rangle}],\mathtt{Y} \rightarrow [\bot\ \mathtt{World}].\mathtt{List\langle X,World,Y\rangle} \mid \emptyset \quad fType(\mathtt{datum},\mathtt{List\langle X,World,Z\rangle}) = \mathtt{X}}{\emptyset;\Gamma \vdash \mathtt{x.datum} : \mathtt{X} \mid \mathtt{X} \rightarrow [\bot\ \mathtt{Book\langle World\rangle}]}\ \text{(T-Field)} \quad \dfrac{\emptyset,\mathtt{X} \rightarrow [\bot\ \mathtt{Book\langle World\rangle}] \vdash \mathtt{X} <: \mathtt{Book\langle World\rangle} \quad \emptyset \vdash \mathtt{X} \rightarrow [\bot\ \mathtt{Book\langle World\rangle}]\ \text{OK} \quad \emptyset \vdash \mathtt{Book\langle World\rangle}\ \text{OK}}{}}{\emptyset;\Gamma \vdash \mathtt{x.datum} : \mathtt{Book\langle World\rangle} \mid \emptyset}\ \text{(T-Subs)}$$

Figure 7: Example derivation.

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

$$\frac{\Delta \vdash \mathtt{T},\overline{\mathtt{T}}\ \text{OK} \quad \mathtt{class\ C\langle \overline{X...}\rangle} \lhd \mathtt{N\ \{...\}} \quad \Delta;\overline{\mathtt{x:T}},\ \mathtt{this:}\exists\emptyset.\mathtt{C\langle \overline{X}\rangle} \vdash \mathtt{e} : \mathtt{T} \mid \emptyset \quad override(\mathtt{m},\mathtt{N},\overline{\mathtt{T}} \rightarrow \mathtt{T})}{\Delta \vdash \mathtt{T\ m(\overline{T\ x})\ \{return\ e\}}\ \text{OK in C}}\ \text{(T-Method)}$$

$$\frac{mType(\mathtt{m},\mathtt{N}) = \overline{\mathtt{T}} \rightarrow \mathtt{T}}{override(\mathtt{m},\mathtt{N},\overline{\mathtt{T}} \rightarrow \mathtt{T})}\ \text{(T-Override)} \qquad \frac{mType(\mathtt{m},\mathtt{N}) \quad undefined}{override(\mathtt{m},\mathtt{N},\overline{\mathtt{T}} \rightarrow \mathtt{T})}\ \text{(T-OverrideUndef)}$$

Class typing: $\boxed{\vdash \mathtt{Q}\ \text{OK}}$

$$\frac{\Delta = \overline{\mathtt{X} \rightarrow [\bot\ \mathtt{T'}]},\ \mathtt{Owner} \rightarrow [\bot\ \mathtt{T}_O],\ \mathtt{This} \rightarrow [\bot\ \exists\emptyset.\mathtt{Owner}],\ \overline{\mathtt{O} \rightarrow [\exists\emptyset.\mathtt{Owner}\ \mathtt{T}]} \quad \emptyset \vdash \Delta\ \text{OK} \quad \Delta \vdash \mathtt{N},\overline{\mathtt{T}}\ \text{OK} \quad \Delta \vdash \overline{\mathtt{M}}\ \text{OK in C} \quad \Delta \vdash \mathtt{N} <: \mathtt{Object\langle Owner,\ This\rangle} \quad \Delta \vdash \mathtt{T}_O <: \exists\emptyset.\mathtt{World\langle\rangle}}{\vdash \mathtt{class\ C\langle \overline{O \lhd T},\ \overline{X \lhd T'},\ Owner \lhd T}_O,\ \mathtt{This} \lhd \exists\emptyset.\mathtt{Owner\rangle} \lhd \mathtt{N\ \{\overline{T\ f};\ \overline{M}\}}\ \text{OK}}\ \text{(T-Class)}$$

Figure 8: WOGJ class and method typing rules.

$$\frac{\forall \iota \rightarrow \{\mathtt{C\langle \overline{T},\ T}_{Owner},\ \mathtt{This}_\iota\mathtt{\rangle;\overline{f \rightarrow v}}\} \in \mathcal{H} : \quad \Delta,\mathtt{This}_\iota \rightarrow [\bot\ \mathtt{T}_{Owner}] \vdash \mathtt{C\langle \overline{T}\rangle}\ \text{OK} \quad \overline{fType(\mathtt{f},\mathtt{C\langle \overline{T}\rangle}) = \mathtt{T'}} \quad \Delta;\mathcal{H} \vdash \overline{\mathtt{v} : \mathtt{T'} \mid \emptyset} \quad \forall \mathtt{v} \in \overline{\mathtt{v}} : \mathtt{v} \neq \mathtt{null} \Rightarrow \mathtt{v} \in dom(\mathcal{H}) \wedge \Delta;\mathcal{H} \vdash \exists\emptyset.\mathtt{This}_\iota <: owner_{\mathcal{H}}(\mathtt{v})}{\Delta \vdash \mathcal{H}\ \text{OK}}\ \text{(F-Heap)}$$

$$\frac{\Delta \vdash \mathcal{H}\ \text{OK} \quad \forall \iota \in fv(\mathtt{e}) : \iota \in dom(\mathcal{H})}{\Delta;\mathcal{H} \vdash \mathtt{e}\ \text{OK}}\ \text{(F-Config)}$$

Figure 9: WOGJ well-formed heaps and configurations.

outside the actual parameter in `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 $\texttt{This}_\iota$ is a subtype of the owner of all values referenced by ι; which ensures that ι 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, ι, in the heap, we add the assumption $\texttt{This}_\iota \rightarrow [\bot\ \texttt{T}_{Owner}]$ to the type environment, where $\texttt{T}_{Owner}$ is the actual type parameter in the owner position of ι'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$\exists$ [7], and for owners-as-dominators of Jo$\exists$ and OGJ [24].

> **Conjecture — Progress** For any Δ, $\mathcal{H}$, $\texttt{e}$, $\texttt{T}$, if $\emptyset; \mathcal{H} \vdash \texttt{e} : \texttt{T} \mid \Delta$ then either there exists $\texttt{e}'$ and $\mathcal{H}'$ where $\mathcal{H}; \texttt{e} \rightsquigarrow \mathcal{H}'; \texttt{e}'$ or there exists $\texttt{v}$ where $\texttt{e} = \texttt{v}$.

> **Conjecture — Preservation** For any $\mathcal{H}$, $\mathcal{H}'$, $\texttt{e}$, $\texttt{e}'$, $\texttt{T}$, if $\emptyset; \mathcal{H} \vdash \texttt{e} : \texttt{T} \mid \emptyset$ and $\mathcal{H}; \texttt{e} \rightsquigarrow \mathcal{H}'; \texttt{e}'$ then $\emptyset; \mathcal{H} \vdash \texttt{e}' : \texttt{T} \mid \emptyset$

> **Conjecture — Owners-as-dominators** For any Δ, $\mathcal{H}$, if $\Delta \vdash \mathcal{H}$ OK then $\forall \iota \rightarrow \{\texttt{N}; \{\overline{\texttt{f} \rightarrow \texttt{v}}\}\} \in \mathcal{H}, \forall \texttt{v}_i \in \overline{\texttt{v}} : \texttt{v}_i \neq \texttt{null} \Rightarrow \Delta; \mathcal{H} \vdash \texttt{This}_\iota <: owner_{\mathcal{H}}(\texttt{v}_i)$.

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 Postodoctoral fellowship.

6. REFERENCES

[1] Jonathan Aldrich and Craig Chambers. Ownership Domains: Separating Aliasing Policy from Mechanism. In *ECOOP*, 2004.

[2] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias Annotations for Program Understanding. In *OOPSLA*, 2002.

[3] 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.

[4] Chandrasekhar Boyapati, Robert Lee, and Martin C. Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *OOPSLA*, 2002.

[5] Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership Types for Object Encapsulation. In *Principles of Programming Languages (POPL)*, 2003.

[6] Chandrasekhar Boyapati and Martin Rinard. A Parameterized Type System for Race-free Java Programs. In *OOPSLA*, 2001.

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

[8] Nicholas Cameron, Sophia Drossopoulou, and Erik Ernst. A Model for Java with Wildcards. In *ECOOP*, 2008.

[9] Nicholas Cameron, Sophia Drossopoulou, James Noble, and Matthew Smith. Multiple Ownership. In *OOPSLA*, 2007.

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

[11] David Clarke, Michael Richmond, and James Noble. Saving the world from bad beans: deployment-time confinement checking. In *OOPSLA*, pages 374–387, 2003.

[12] David G. Clarke, John M. Potter, and James Noble.

Ownership Types for Flexible Alias Protection. In *OOPSLA*, 1998.

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

[14] Werner Dietl, Sophia Drossopoulou, and Peter Müller. Generic Universe Types. In *ECOOP*, 2007.

[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 *ECOOP*, 2006.

[19] Mads Torgersen and Erik Ernst and Christian Plesner Hansen. Wild FJ. In *Foundations of Object-Oriented Languages (FOOL)*, 2005.

[20] P. Müller and A. Poetzsch-Heffter. Universes: A Type System for Controlling Representation Exposure. In *Programming Languages and Fundamentals of Programming*, 1999.

[21] James Noble, Jan Vitek, and John Potter. Flexible Alias Protection. In *ECOOP*, 1998.

[22] Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. Featherweight Generic Ownership. In *Formal Techniques for Java-like Programs (FTfJP)*, 2005.

[23] Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. Featherweight generic confinement. *J. Funct. Program.*, 16(6), 2006.

[24] Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. Generic Ownership for Generic Java. In *OOPSLA*, 2006.

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

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

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

[28] Tobias Wrigstad, Filip Pizlo, Fadi Meawad, Lei Zhao, and Jan Vitek. Loci: Simple thread-locality for Java. In *ECOOP*, 2009. To Appear.

[29] Tian Zhao, Jason Baker, James Hunt, James Noble, and Jan Vitek. Implicit ownership types for memory management. *Sci. Comput. Program.*, 71(3):213–241, 2008.

APPENDIX

A. ELIDED RULES

Congruence rules: $\boxed{\mathtt{e};\mathcal{H} \rightsquigarrow \mathtt{e};\mathcal{H}}$

$$\frac{\mathtt{e};\mathcal{H} \rightsquigarrow \mathtt{e}';\mathcal{H}' \qquad \mathtt{e}' \neq \mathtt{err}}{\mathtt{e.f};\mathcal{H} \rightsquigarrow \mathtt{e}'\mathtt{.f};\mathcal{H}'} \text{ (RC-Field)} \qquad \frac{\mathtt{e}_1;\mathcal{H} \rightsquigarrow \mathtt{e}_1';\mathcal{H}' \qquad \mathtt{e}_1' \neq \mathtt{err}}{\iota\mathtt{e}_1\mathtt{.f = e}_2;\mathcal{H} \rightsquigarrow \iota\mathtt{e}_1'\mathtt{.f = e};\mathcal{H}'} \text{ (RC-Assign-1)} \qquad \frac{\mathtt{e}_2';\mathcal{H} \rightsquigarrow \mathtt{e}_2';\mathcal{H}' \qquad \mathtt{e}_2' \neq \mathtt{err}}{\iota\mathtt{.f = e}_2;\mathcal{H} \rightsquigarrow \iota\mathtt{.f = e}_2';\mathcal{H}'} \text{ (RC-Assign-2)}$$

$$\frac{\mathtt{e};\mathcal{H} \rightsquigarrow \mathtt{e}';\mathcal{H}'}{\mathtt{e.m}(\overline{\mathtt{e}});\mathcal{H} \rightsquigarrow \mathtt{e}'\mathtt{.m}(\overline{\mathtt{e}});\mathcal{H}'} \text{ (RC-Invk-Recv)} \qquad \frac{\mathtt{e}_i;\mathcal{H} \rightsquigarrow \mathtt{e}_i';\mathcal{H}' \qquad \mathtt{e}_i' \neq \mathtt{err}}{\iota\mathtt{.m}(\overline{\mathtt{v}}\mathtt{,}\ \mathtt{e}_i\mathtt{,}\ \overline{\mathtt{e}});\mathcal{H} \rightsquigarrow \iota\mathtt{.m}(\overline{\mathtt{v}}\mathtt{,}\ \mathtt{e}_i'\mathtt{,}\ \overline{\mathtt{e}});\mathcal{H}'} \text{ (RC-Invk-Arg)}$$

Null pointer exceptions: $\boxed{\mathtt{e};\mathcal{H} \rightsquigarrow \mathtt{err};\mathcal{H}}$

$$\frac{}{\mathtt{null.f};\mathcal{H} \rightsquigarrow \mathtt{err};\mathcal{H}} \text{ (R-Field-Null)} \qquad \frac{}{\mathtt{null.f = e};\mathcal{H} \rightsquigarrow \mathtt{err};\mathcal{H}} \text{ (R-Assign-Null)} \qquad \frac{}{\mathtt{null.m}(\overline{\mathtt{e}});\mathcal{H} \rightsquigarrow \mathtt{err};\mathcal{H}} \text{ (R-Invk-Null)}$$

Error propogation: $\boxed{\mathtt{e};\mathcal{H} \rightsquigarrow \mathtt{err};\mathcal{H}}$

$$\frac{\mathtt{e};\mathcal{H} \rightsquigarrow \mathtt{err};\mathcal{H}'}{\mathtt{e.f};\mathcal{H} \rightsquigarrow \mathtt{err};\mathcal{H}'} \text{ (RC-Field-Err)} \qquad \frac{\mathtt{e}_1;\mathcal{H} \rightsquigarrow \mathtt{err};\mathcal{H}'}{\mathtt{e}_1\mathtt{.f = e}_2;\mathcal{H} \rightsquigarrow \mathtt{err};\mathcal{H}'} \text{ (RC-Assign-Err-1)} \qquad \frac{\mathtt{e}_2;\mathcal{H} \rightsquigarrow \mathtt{err};\mathcal{H}'}{\iota\mathtt{.f = e}_2;\mathcal{H} \rightsquigarrow \mathtt{err};\mathcal{H}'} \text{ (RC-Assign-Err-2)}$$

$$\frac{\mathtt{e};\mathcal{H} \rightsquigarrow \mathtt{err};\mathcal{H}'}{\mathtt{e.m}(\overline{\mathtt{e}});\mathcal{H} \rightsquigarrow \mathtt{err};\mathcal{H}'} \text{ (RC-Invk-Recv-Err)} \qquad \frac{\mathtt{e}_i;\mathcal{H} \rightsquigarrow \mathtt{err};\mathcal{H}'}{\iota\mathtt{.m}(\overline{\mathtt{v}}\mathtt{,}\ \mathtt{e}_i\mathtt{,}\ \overline{\mathtt{e}});\mathcal{H} \rightsquigarrow \mathtt{err};\mathcal{H}'} \text{ (RC-Invk-Arg-Err)}$$

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

Lookup Functions

$$\frac{}{fields(\mathtt{Object}) = \emptyset} \qquad \frac{\mathtt{class\ C<}\overline{\mathtt{X}\lhd\ \mathtt{T}_u}\mathtt{>} \lhd \mathtt{D<...>\ \{}\overline{\mathtt{U\,f;}}\ \overline{\mathtt{M}}\mathtt{\}} \qquad fields(\mathtt{D}) = \overline{\mathtt{g}}}{fields(\mathtt{C}) = \overline{\mathtt{g}}, \overline{\mathtt{f}}}$$

$$\frac{\mathtt{class\ C<}\overline{\mathtt{X}\lhd\ \mathtt{T}_u}\mathtt{>} \lhd \mathtt{N\ \{}\overline{\mathtt{U\,f;}}\ \overline{\mathtt{M}}\mathtt{\}} \qquad \mathtt{f} \notin \overline{\mathtt{f}}}{fType(\mathtt{f}, \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = fType(\mathtt{f}, [\overline{\mathtt{T}/\mathtt{X}}]\mathtt{N})} \qquad \frac{\mathtt{class\ C<}\overline{\mathtt{X}\lhd\ \mathtt{T}_u}\mathtt{>} \lhd \mathtt{N\ \{}\overline{\mathtt{U\,f;}}\ \overline{\mathtt{M}}\mathtt{\}}}{fType(\mathtt{f}_i, \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = [\overline{\mathtt{T}/\mathtt{X}}]\mathtt{U}_i}$$

$$\frac{\mathtt{class\ C<}\overline{\mathtt{X}\lhd\ \mathtt{T}_u}\mathtt{>} \lhd \mathtt{N\ \{}\overline{\mathtt{U\,f;}}\ \overline{\mathtt{M}}\mathtt{\}} \qquad \mathtt{m} \notin \overline{\mathtt{M}}}{mBody(\mathtt{m}, \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = mBody(\mathtt{m}, [\overline{\mathtt{T}/\mathtt{X}}]\mathtt{N})} \qquad \frac{\mathtt{class\ C<}\overline{\mathtt{X}\lhd\ \mathtt{T}_u}\mathtt{>} \lhd \mathtt{N\ \{}\overline{\mathtt{U'\,f;}}\ \overline{\mathtt{M}}\mathtt{\}} \quad \mathtt{U\,m\,(}\overline{\mathtt{U\,x}}\mathtt{)\ \{return\ e}_0\mathtt{;\}} \in \overline{\mathtt{M}}}{mBody(\mathtt{m}, \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = (\overline{\mathtt{x}}; [\overline{\mathtt{T}/\mathtt{X}}]\mathtt{e}_0)}$$

$$\frac{\mathtt{class\ C<}\overline{\mathtt{X}\lhd\ \mathtt{T}_u}\mathtt{>} \lhd \mathtt{N\ \{}\overline{\mathtt{U\,f;}}\ \overline{\mathtt{M}}\mathtt{\}} \qquad \mathtt{m} \notin \overline{\mathtt{M}}}{mType(\mathtt{m}, \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = mType(\mathtt{m}, [\overline{\mathtt{T}/\mathtt{X}}]\mathtt{N})} \qquad \frac{\mathtt{class\ C<}\overline{\mathtt{X}\lhd\ \mathtt{T}_u}\mathtt{>} \lhd \mathtt{N\ \{}\overline{\mathtt{U'\,f;}}\ \overline{\mathtt{M}}\mathtt{\}} \quad \mathtt{U\,m\,(}\overline{\mathtt{U\,x}}\mathtt{)\ \{return\ e}_0\mathtt{;\}} \in \overline{\mathtt{M}}}{mType(\mathtt{m}, \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = [\overline{\mathtt{T}/\mathtt{X}}](\overline{\mathtt{U}} \to \mathtt{U})}$$

Figure 11: Method and field lookup functions for WOGJ.

$$\frac{\mathcal{H} = \overline{\iota \rightarrow \mathtt{\{C<}\overline{\mathtt{T}}\mathtt{,\ T}_{Owner}\mathtt{,\ This}_\iota\mathtt{>;\ ...\}}} \qquad \Delta, \overline{\mathtt{This}_\iota \rightarrow [\bot\ \mathtt{T}_{Owner}]} \vdash \mathtt{T} <: \mathtt{T}'}{\Delta;\mathcal{H} \vdash \mathtt{T} <: \mathtt{T}'} \text{ (H-S)} \qquad \frac{\mathcal{H} = \overline{\iota \rightarrow \mathtt{\{C<}\overline{\mathtt{T}}\mathtt{,\ T}_{Owner}\mathtt{,\ This}_\iota\mathtt{>;\ ...\}}} \qquad \Delta, \overline{\mathtt{This}_\iota \rightarrow [\bot\ \mathtt{T}_{Owner}]}; \overline{\iota:\mathtt{R}}, \Gamma \vdash \mathtt{e} : \mathtt{T} \mid \Delta'}{\Delta;\mathcal{H}, \Gamma \vdash \mathtt{e} : \mathtt{T} \mid \Delta'} \text{ (H-T)}$$

Figure 12: Using the heap as an environment in WOGJ.