Encoding Ownership Types in Java

Nicholas Cameron and James Noble

Victoria University of Wellington, New Zealand

Abstract. Ownership types systems organise the heap into a hierarchy which can be used to support encapsulation properties, effects, and invariants. Ownership types have many applications including parallelisation, concurrency, memory management, and security. In this paper, we show that several flavours and extensions of ownership types can be entirely encoded using the standard Java type system.
Ownership types systems usually require a sizable effort to implement and the relation of ownership types to standard type systems is poorly understood. Our encoding demonstrates the connection between ownership types and parametric and existential types. We formalise our encoding using a model for Java's type system, and prove that it is sound and enforces an ownership hierarchy. Finally, we leverage our encoding to produce lightweight compilers for Ownership Types and Universe Types — each compiler took only one day to implement.

1 Introduction

Ownership types describe the topology of the heap in the program source code. They come in several varieties (context-parametric [16], Universes [17], Ownership Domains [3], OGJ [27], and more) and have many practical applications, including preventing data races [6,18], parallelisation [15,5], real-time memory management [4], and enforcing architectural constraints [2].

Ownership types usually require large, complicated type systems and compilers, and their relation to standard type theory is not well understood. We give a simple encoding from ownership types to standard generic Java by extending the previously identified relationship between ownership types and parametric types [26,27]. This previous work encoded ownership parameters as type parameters, but treated the current object's ownership context (the `this` or `This` context) specially; we treat it as a standard type parameter, hidden externally by existential quantification [13]. With this technique we can encode ownership types (with generics and existential quantification), Ownership Domains, and Generic Universe Types. Furthermore, by unpacking the `This` parameter we can support a range of extensions, including inner classes [7], dynamic aliases [15], fields as contexts [12], and existential downcasting [31], within the same standard type system.

Contributions and Organisation The contributions of this paper are: a thorough discussion of how various flavours and extensions of ownership types can be

encoded in a standard type system, such as Java's (Sect. 3), a formal type system which captures these concepts (including variations and extensions) and a soundness proof for this system which demonstrates that our encoding enforces the ownership hierarchy (Sect. 4), and compilers for Generic Universe Types and Ownership Types (Sect. 5).

Our work is of benefit to theoreticians and implementors: it provides an element of the fundamental underpinnings of ownership types and a shorter path to the implementation of languages with ownership types. We do not introduce new features or make existing approaches more expressive. We do not envisage that programmers would use a language like our encoding directly.

Additionally, we give background on Java generics and ownership types in Sect. 2 and conclude and describe future work in Sect. 6.

2 Background

In this section, we describe ownership types and features of the Java type system used in our encoding.

2.1 Java Generics and Wildcards

Java has featured parametric and existential types since version 5.0, in the form of *generics* and *wildcards* [20]. Java types consist of a class name and a (possibly empty) list of actual type parameters, for example, we can describe a list of books as `List<Book>`, this requires a class (or interface) with formal type parameters, e.g., `class List<X> {...}`. The formal type parameters (e.g., `X`) may be used in the body of the class; outside the class body they must be instantiated with actual parameters (e.g., `Book`).

Generic types must be *invariant* with respect to subtyping. However, it is sometimes safe and desirable to make generic types co- or contravariant. To support this, Java has wildcards [28]: an object of type `List<? extends Book>` is a covariant list of books, that is, a list of some subtype of book. To remain sound, covariant lists must be read-only and contravariant lists (indicated by lower bounds, using the `super` keyword) write-only; wildcards enforce this. Formal models of Java typically use bounded existential types to represent wildcards [11]: our covariant list is denoted $\exists$`X`$\rightarrow$`[`$\bot$ `Book].List<X>` (we use `[L U]` to denote lower and upper bounds on type variables; $\bot$, the bottom type, indicates no lower bound).

A wildcard *hides* a type parameter; for example, we can store (due to subtyping) an object of type `List<Book>` in a variable of type `List<?>`: the wildcard hides the witness type `Book`. Java does allow the type to be temporarily named, but only as a fresh type variable, this is known as *wildcard capture* and corresponds to *unpacking* an existential type[1]. For example, `List<?>` can be *capture converted* to `List<Z>`, where `Z` is fresh; however, the type system does not know of any relationship between `Z` and `Book`.

[1] Subtyping of concrete types to wildcard types corresponds to *packing*.

2.2 Ownership Types

At their most abstract, ownership types [16] are a mechanism for organising the heap into a hierarchy of contexts. The type system ensures that objects' positions in the hierarchy are reflected in their types. This soundness property allows contexts to be used to specify encapsulation properties (for which ownership types are famous), such as owners-as-dominators [16] and owners-as-modifiers [17], or to specify effects [15] or invariants [24]. Several mechanisms for reflecting the ownership hierarchy in types have been proposed; these can be separated into parameter-based systems, where types are parameterised by contexts (such as 'vanilla' ownership types [16,15], multiple ownership [12], and ownership domains [3]) and annotation-based systems, where types are annotated to describe relative position in the hierarchy (such as Universes [17]).

There have been several syntactic (but semantically equivalent) variations in the way ownership types are denoted, in our source language we prefix an object's type with its owner and parameterise it with actual context parameters. A class is declared without an explicit owner (only context parameters) and the `owner` keyword is added to the language for use as an actual context parameter (similarly to the `this` keyword); for example:

```
class List<d> {
    this:Node<d> first;
}
class Node<d> {
    owner:Node<d> next;
    d:Object datum;
}
```

Ownership types

Here, a list object owns all of its nodes and the context parameter `d` holds the data in the list. We will use this list as a running example.

Encapsulation and Effects Most ownership systems consist of a descriptive part (describing the topology of the heap) and a prescriptive part, which uses the described topology to specify an encapsulation policy or effect system. Encapsulation properties can restrict aliasing (e.g., owners-as-dominators, associated with vanilla ownership types [16]) or access (e.g., owners-as-modifiers, from Universes [17]). An effect system describes how objects are accessed, rather than restricting access. In this paper we concentrate on the descriptive aspects of ownership and so we will not describe these policies in detail.

Universes Universes [17] are an annotation-based ownership system. Types may be annotated with `rep` (denoting that objects of this type are owned by `this`), `peer` (objects are in the same context as `this`), or `any` (objects are in an unknown context). Generic Universe Types [19] support both type parametricity and universe modifiers; the programmer can write types such as `rep List<peer Book>`, which represents a list (owned by the current object) of books in the current context. Universe types and ownership types describe the same hierarchies [9]. Uni-

verse types are simpler to write than ownership types, but less expressive. The above list example is expressed using Universes below, the data in the list can be described more precisely (as in ownership types) if we were to use generics.

```
class List {
    rep Node first;
}
class Node {
    peer Node next;
    any Object datum;
}
```

Universe

2.3 OGJ

Ownership types and generics can be combined in an orthogonal fashion [19,10], giving the benefits and flexibility of both systems. They can also be integrated, as in Ownership Generics Java (OGJ [27]); the benefits of both systems are still gained, but with only a single kind of parameter: type parameters are used to represent context parameters. The only extra ingredient in OGJ (beyond standard Java generics) is a `This` type parameter which represents not a type, but the current context. This type parameter is treated specially by the formal type rules.

Our list example can be written in OGJ:

```
class List<D, Owner> {
    Node<D, This> first;
}
class Node<D, Owner> {
    Node<D, Owner> next;
    Object<D> datum;
}
```

OGJ

The syntax is almost identical to the standard ownership types version, other than the owner of a type is specified as the last type parameter. The semantics, however, are different: all parameters are treated as type parameters by the type system, the usual rules for type checking Java are applied, rather than special ownership types rules. The exception is in dealing with the `This` owner of `first`, here, special rules must be applied.

Featherweight Generic Confinement (FGC [26]) uses the same representation of contexts as type parameters, but without any support for the `This` context. The result is encapsulation within static packages, but not within dynamically allocated objects.

3 Encoding Ownership Types into Java

In this section we describe how we encode source ownership types programs into Java. As in FGC [26] and OGJ [27], we represent the owner of a class and its context parameters with type parameters. Actual context parameters are encoded as actual type parameters. We create a formal type parameter to represent the `this` context [13], bounded above by `Owner`. The inside relation (context ordering) is encoded by subtyping (as in OGJ). Since `this` cannot be named outside its class declaration, we must hide the corresponding `This` type parameter in types, which is done using Java wildcards; conveniently, the wildcard will inherit the bound declared on `This`. Our basic ownership types list example (Sect. 2.2) is encoded as:

```
class List<D, Owner extends World, This extends Owner> {
    Node<D, This, ?> first;
}
class Node<D, Owner extends World, This extends Owner> {
    Node<D, Owner, ?> next;
    Object<D, ?> datum;
}
```

Java

Actual context parameters are either `World` (which represents the root context) or formal context variables (either quantified or with class scope). The inherited or explicit bounds on these type variables produce a partial ordering on type parameters corresponding to the ownership hierarchy[2]. Because there are no concrete types representing contexts (other than `World`), the hierarchy is an illusion: an omniscient type checker would know that all context-type variables ultimately hold `World`. The opacity of existential types ensures that the illusory hierarchy is respected during type checking.

Type systems must treat existentially quantified variables as hiding unique types; this gives the correct behaviour for ownership types in our encoding by treating each `This` context as unique. If we did not always hide the `This` parameter, ownership typing would not be effective[3]:

```
List<World, World, X> l1 = new List<World, World, X>();
List<World, World, X> l2 = new List<World, World, X>();
l1.first = l2.first;        //OK, but should be an error
```

Java

Universes Generic Universe Types can be encoded into ownership types [9], and then into Java using the above scheme. The only obstacle is that the universe modifier `any` corresponds to an existentially quantified owner (see below); `any`

[2] There are effectively two subtype hierarchies: one of real objects with `Object` at its root, and one of ownership contexts with `World` at its root.

[3] In this section we will use wildcards in `new` expressions, this is not allowed in Java and we describe how to avoid this in Sect. 5.

can be encoded as an unbounded wildcard. The translation of the Universes basic list is given below. It is simpler than the ownership types version because we do not need to encode the context parameter; note the owner of `datum` is a wildcard, which encodes `any`.

```
class List<Owner extends World, This extends Owner> {
    Node<This, ?> first;
}
class Node<Owner extends World, This extends Owner> {
    Node<Owner, ?> next;
    Object<?, ?> datum;
}
```
Java

The extension to Generic Universe Types is straightforward, type parameters remain in the encoding, upper bounds are encoded in the same way as other types.

Ownership Domains Ownership domains [3] support more flexible topologies and a more flexible encapsulation property than ownership types. Topologically, ownership domains allow for multiple contexts (called domains) per object; objects can belong to any of these contexts and all contexts are nested within the object's owner.

To support multiple contexts per object in our encoding we allow multiple parameters in place of the single `This` parameter. All these parameters are given the upper bound of `Owner` and all must be hidden with wildcards to create the phantom ownership hierarchy. Types are encoded in the same way as for ownership types.

For example, the following class with two domains and a single domain parameter,

```
class C<domP> {  domain dom1, dom2;  }
```
ODs

it is encoded as the Java class,

```
class C<DomP, Owner, Dom1 extends Owner, Dom2 extends Owner> {}
```
Java

3.1 Extensions to Ownership Types

There has been much work on making ownership types systems more descriptive and more flexible. Generally the underlying ownership hierarchy is unchanged, but it can be described more precisely in the source code, usually combined with a relaxation of encapsulation properties in certain circumstances. In this section we describe several extensions to ownership types and how they can be encoded.

Bounds Context parameters may be given upper and lower bounds [15,10] with respect to the ownership hierarchy. These are usually denoted `inside` and `outside`, respectively. For example, `class C<a outside owner, b inside a>`.

Upper bounds on context parameters can easily be replicated using upper bounds on the corresponding type parameters (e.g. `B extends A`). The encoded bounds are with respect to the subtype hierarchy, within which the ownership hierarchy is encoded. Lower bounds cannot be encoded in Java without changing the type system to support lower bounds on type parameters.

Context Parametric Methods Methods may be parameterised by contexts [14,30] in an ownership system in the same way as they can be parameterised by types in Java. This allows for better code reuse. For example:

```
<a,b> a:Node<b> next(a:Node<b> n) {
    return n.next;
}
```

Ownership types

This method will work for all possible nodes; without context-parametric methods, such a method could not be written.

Context parametric methods are easily encoded as type parametric Java methods, upper bounds on context parameters can be handled as above:

```
<A,B> Node<B, A, ?> next(Node<B, A, ?> n) {
    return n.next;
}
```

Java

Inner Classes Ownership types systems can be made more flexible by giving inner classes access to the `this` and `owner` parameters of the surrounding class [7]. This increases the descriptiveness of the type system because more contexts can be named inside an inner class. Owners-as-dominators can be relaxed to allow instantiations of inner classes to hold references to their surrounding objects (e.g., `curNode` field in the following example). This allows iterators to be implemented in an owners-as-dominators system, an early obstacle to acceptance of ownership types systems. We extend our list example:

```
class List<d> {
    ...
    class Iterator {
        List.this:Node<d> curNode;
        d:Object next() {
            d:Object val = curNode.datum;
            curNode = curNode.next();
            return val;
        }
    }
}

class Client {
    void m(this:List<world> l) {
        this:Iterator i = l.new this:Iterator()
        world:Object first = i.next();
    }
}
```

Ownership types

Inner classes must be able to name the context of their surrounding class; this happens naturally in Java, an inner class can name type parameters of its surrounding class. We must be careful to avoid hiding the generated type parameter by adding `This` parameters for both inner and outer classes. This is easily accomplished by prepending the name of the class to the names of the `Owner` and `This` parameters (we elide some bounds):

```
class List<D, Owner, This extends Owner> {
    ...
    class Iterator<It_Owner, It_This extends It_Owner> {
        Node<D, This, ?> curNode;
        Object<D, ?> next() {
            Object<D, ?> val = curNode.datum;
            curNode = curNode.next();
            return val;
        }
    }
}

class Client<Owner, This extends Owner> {
    void m(List<World, This, ?> l) {
        Iterator<This, ?> i = l.new Iterator<This, ?>();
        Object<World, ?> first = i.next();
    }
}
```

Java

Dynamic Aliases An alternative solution to the iterators problem under owners-as-dominators is to allow *dynamic aliases* [15], that is allow variables on the stack to reference objects which break owners-as-dominators, and only enforce owners-as-dominators on the heap. Dynamic aliases achieve this by allowing local variables to be used as contexts. Extending the original list example:

```
class Iterator<d> {
    owner:Node<d> curNode;
    d:Object next() {
        d:Object val = curNode.datum;
        curNode = curNode.next();
        return val;
    }
}

class Client {
    void m(final this:List<world> l) {
        l:Iterator<world> i = new l:Iterator<world>();
        world:Object first = i.next();
    }
}
```

Ownership types

The variable `l` cannot be named outside of `m`, and so the dynamic alias to `i` (owned by `l`) cannot be stored in the heap. It is only sound to use final variables to name contexts.

An object's context is represented by its hidden `This` argument; therefore, encoding dynamic aliases in Java requires naming that argument using a fresh, temporary type variable which is introduced as an extra type parameter to a method. Unpacking the hidden `This` argument to the named variable is achieved by wildcard capture:

```
class Iterator<D, Owner extends World, This extends Owner> {
    Node<D, Owner, ?> curNode;
    Object<D, ?> next() {
        Object<D, ?> val = curNode.datum;
        curNode = curNode.next();
        return val;
    }
}

class Client<Owner extends World, This extends Owner> {
    void m(List<World, This, ?> l) {
        this.mAux(l)
    }

    <L> void mAux(List<World, This, L> l) {
        Iterator<World, L, ?> i = new Iterator<World, L, ?>;
        Object<World, ?> first = i.next();
    }
}
```
Java

The wildcard which hides `l`'s `This` argument is capture converted to the fresh type variable `L` when `mAux` is called. Using `l` as an owner in the source program is encoded to using `L` as an owner. `L` can only be named within the scope of `mAux`, and this corresponds to the scope of `l`.

Our example is simple because it does not require other state to be passed to `mAux`. In a more realistic example we would need to pass any data accessed in `m` to `mAux`, and back again if it is not passed by reference. A simpler encoding is to modify the original method so that the `This` argument of `l` is captured by calling `m` (rather than when `mAux`). The simpler encoding only works if the variable being used as a context is an argument rather than a local variable. Note that the call-sites of `m` do not have to be modified, despite the extra type parameter, due to Java's type parameter inference:

```
class Client<Owner, This> {
    <L> void m(List<World, This, L> l) { ... }  //body as mAux
}
```
Java

Fields as Contexts Similarly to local variables, final fields can be used to name contexts [12], this again improves flexibility. We can extend the list example:

```
class List<d> {
    final this:Node<d> first;
    first:Object f2;    //owned by a field
}
```
Ownership types

Paths of final fields may also be used as contexts [12], e.g., one could allow the type `f3.first:Object`, where `f3` is a final field of type `List`.

We encode fields used as contexts by adding their hidden `This` parameters to the class's parameter list:

```
class List<D, Owner extends World, This extends Owner,
           First extends This> {
    final Node<D, This, ? extends First> first;
    Object<First, ?> f2;
}
```

Java

Instantiating this class requires that the value of first is passed into the constructor, wildcard capture is used to name `First` and then both `this` and `First` are hidden by wildcards.

Existential Quantification Just as type variables may be quantified existentially, so may context variables [10]. This gives existential ownership types such as ∃`o.o:Object` or ∃`o.this:List<o>`. Such quantification has two benefits: context variance, that is subtyping which is variant with respect to the ownership hierarchy, and expressing partial knowledge about contexts (e.g., an unknown context or some unknown context within another known context). Existential quantification is the mechanism which underlies a number of proposals involving some kind of variance annotations on contexts [23,8].

Existentially quantified contexts can be encoded as wildcards. Since wildcards are syntactic sugar for existential types, this is not surprising. Both upper and lower bounds can be straightforwardly encoded. The only difficulty is if quantified contexts have both upper and lower bounds, which is not supported by Java wildcards. This should not be a problem, however, because quantification is usually provided by variance annotations or wildcard-like syntax.

Existential Downcasting Downcasting is a common feature in programs, especially those that do not use generics. When downcasting from type `A` to type `B`, if `B` has context parameters which `A` does not, these must be synthesised. Wrigstad and Clarke propose the use of "existential owners" to handle these introduced context parameters [31]. For example:

```
    void m(this:Object x) {
        this:List<d> l = (this:List<d>) x;
        d:Object first = l.first.datum;
        l.first.datum = new d:Object();
    }
```

Ownership types

Here `x` is cast from type `this:Object` to `this:List<d>`, the `d` context is a fresh context (an "existential owner") that can be named in the scope of the method and allows operations on `l` to take place. Objects owned by `d` cannot be stored in the heap, outside of the original data structure, since `d` can only be

named locally. Note that there is no explicit quantification, although "existential owners" correspond to unpacked context-existential types [8].

We can cast x to a type where `D` is hidden by a wildcard, although we cannot cast directly to a type containing `D` because `D` is not in scope. We must split the method in order to name `D` using capture conversion:

```
void m(Object<This, ?> x) {
    this.mAux((List<?, This, ?>) x);
}
<D> void mAux(List<D, This, ?> l) {
    Object<D, ?> first = l.first.datum;
    l.first.datum = new Object<D, ?>();
}
```

Java

Owners-as-Dominators The owners-as-dominators property specifies that all reference paths from the root of the ownership hierarchy to any object pass through that object's owner: owners dominate reference paths. The property is enforced by restricting which contexts can be named: if only contexts outside the current context can be named, then no references can exist *into* contexts other than the one owned by the current `this` object.

We have previously sketched how owners-as-dominators can be supported in an encoding of ownership into Java [13]. This approach can be duplicated here with the same drawback: owners-as-dominators can only be guaranteed if the Java compiler is modified, it cannot be supported as a pre-processor step like the rest of the encodings discussed. The modifications are not major: a small change to the well-formedness rules for classes and types to ensure that context parameters are outside the declared owner (the usual requirement for ownership types to support owners-as-dominators). The issue is that at intermediate steps of computation the compiler might allow the `This` parameter to be named in types: this is not a problem for descriptive ownership because it is only temporary, but it can allow owners-as-dominators to be violated.

4 Formalisation

To be sure that our encoding of ownership types in Java is sound, we have formalised it in Java's type system. This formalisation (Tame FJ_{Own}) follows the approach of OGJ [27], in representing context parameters as type parameters, but, by supporting existential types, we do not need any special machinery to deal with ownership issues.

The bulk of the formal system is relatively standard or follows Tame FJ [11]. Differences from Tame FJ to support ownership are highlighted in grey. Tame FJ_{Own} extends Tame FJ, but we are still modelling the Java type system without extension: the additions to Tame FJ are either a convenience (syntactic separation of context and type parameters), or used only for proving soundness (locations as run-time owners and $\star$ in object creation).

We also add field assignment and `null` and a heap and casting (to model dynamic downcasts), and make some small improvements elsewhere; these changes are not highlighted. For the sake of brevity, we do not describe the parts unchanged from Tame FJ. Parts of the operational semantics, well-formed environments and heaps, auxiliary functions, and rules for using the heap as an environment are available in an accompanying technical report [1].

e	::=	γ | `null` | e`.f` | e`.f =` e | e`.<`$\overline{P}$`,` $\overline{\mathcal{P}}$`>m(`$\overline{e}$`)`	*expressions*
		| `new C<`$\overline{\mathcal{T}}$`,`$\star$`>` | `(`T`)`e	
v	::=	ι | `null`	*values*
Q	::=	`class C<`$\overline{\texttt{X}\lhd T}$`,` $\overline{\texttt{O}\lhd \tau}$`,Owner`$\lhd\tau$`, This`$\lhd\tau$`>` $\lhd$ N $\{\overline{T\ \texttt{f};}\ \overline{M}\}$	
M	::=	`<`$\overline{\texttt{X}\lhd T}$`,` $\overline{\texttt{O}\lhd \mathcal{T}}$`>` T `m(`$\overline{T\ \texttt{x}}$`)` $\{$`return` e`;`$\}$	*method declarations*
N	::=	`C<`$\overline{T}$`,` $\overline{\tau}$`>` | `Object<`τ`,`τ`>`	*class types*
R	::=	N | `X`	*non-existential types*

T, U	::=	$\exists\Delta.N$ | $\exists\emptyset$`.X`	*types*	$\mathcal{T}$	::=	T | τ	*types and contexts*
P	::=	T | $\star$	*method type parameters*	$\mathcal{P}$	::=	$\mathcal{T}$ | $\star$	*method parameters*
$\mathcal{X}$,$\mathcal{Y}$	::=	`X` | `O` | v	*type parameters*	τ	::=	`World<>` | `O` | v	*contexts*

Δ	::=	$\overline{\mathcal{X}\rightarrow\texttt{[}B_l\ B_u\texttt{]}}$	*type environments*	`x, this`	*variables*
B	::=	$\mathcal{T}$ | $\bot$	*bounds*	`X, Y`	*type variables*
Γ	::=	$\overline{\gamma{:}T}$	*variable environments*	`O, Owner, This`	*context variables*
γ	::=	ι | `x`	*locations or variables*	ι	*locations*
$\mathcal{H}$	::=	$\overline{\iota\rightarrow\{N;\ \overline{\texttt{f}\rightarrow v}\}}$	*heaps*	`C, Object, World`	*class names*
				`f, g`	*field names*
				`m`	*method names*

Fig. 1. Syntax of Tame FJ$_{Own}$.

Syntax The syntax of Tame FJ$_{Own}$ is given in Fig. 1. For convenience, and following OGJ [27], we syntactically separate types and type parameters used to represent contexts from regular types: we use τ to denote types which represent contexts, T to denote regular types, and $\mathcal{T}$ to denote either type; likewise for parameters, we use `O` to denote type parameters which represent context parameters, `X` for regular type parameters, and $\mathcal{X}$ for either kind. Importantly, the two kinds of type are treated almost identically by the type system. We could do without this convenience by examining the type's top supertype: contexts will be bounded by `World`, other types by `Object`. Type parameters for method invocations must be fully specified. We use $\star$ to indicate that the type parameter

should inferred; this allows us to model wildcard capture. We also use $\star$ in object creation.

We allow values (v, that is addresses, and `null`, which corresponds to `World`) to be context (and thus type) parameters (only at runtime) in order to prove a stronger soundness property (see Sect. 4.1). Values are not allowed as parameters in source code.

We use a few shorthands for types: `C` for `C<>`, and R for $\exists\emptyset.R$.

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

$$\frac{\mathcal{X} \in \Delta}{\Delta \vdash \mathcal{X} \text{ OK}}\ \text{(F-Var)} \qquad \frac{}{\Delta \vdash \texttt{World<>} \text{ OK}}\ \text{(F-World)} \qquad \frac{}{\Delta \vdash \bot \text{ OK}}\ \text{(F-Bottom)} \qquad \frac{}{\Delta \vdash \star \text{ OK}}\ \text{(F-Star)} \qquad \frac{\Delta \vdash \Delta' \text{ OK} \quad \Delta, \Delta' \vdash \texttt{N} \text{ OK}}{\Delta \vdash \exists \Delta'.\texttt{N} \text{ OK}}\ \text{(F-Exists)}$$

$$\frac{\begin{array}{c}\Delta \vdash \overline{T}, \overline{\tau}, \tau_o \text{ OK} \\ \overline{\mathcal{T}} = \overline{T}, \overline{\tau}, \tau_o, \tau_t \qquad \Delta(\tau_t) = [\bot\ \mathcal{T}] \\ \texttt{class C<}\overline{\mathcal{X} \lhd \mathcal{T}_u}\texttt{>} \lhd N\{\ldots\} \\ \Delta \vdash \overline{\mathcal{T}} <: [\overline{\mathcal{T}/\mathcal{X}}]\mathcal{T}_u\end{array}}{\Delta \vdash \texttt{C<}\overline{T}\texttt{>} \text{ OK}}\ \text{(F-Class)} \qquad \frac{\Delta \vdash \tau_o \text{ OK} \quad \Delta(\tau_t) = [\bot\ \mathcal{T}]}{\Delta \vdash \texttt{Object<}\tau_o\texttt{,}\tau_t\texttt{>} \text{ OK}}\ \text{(F-Object)}$$

Fig. 2. Tame FJ$_{Own}$ well-formed types, type environments, and heaps.

Well-formed Types Well-formed types are defined in Fig. 2. In F-Class and F-Object, we do not check that the type parameter in the `This` position is well-formed. Instead we check that it is in the environment and is bounded below by bottom. This ensures that it is always an in-scope variable (in fact it is usually a quantified variable, although this does not need to be enforced) and that no other type can be derived to be a subtype of it (as would be the case if it had a lower bound). This ensures that the `This` context cannot be named by using subsumption.

Type Checking Selected type rules are given in Fig. 3. So that fields owned by `This` can be initialised, object creation (T-New) does not take any (value) parameters (i.e., we don't have constructors, at runtime all fields are initialised to `null`). This requires `null` and the T-Null rule. The actual type parameter in the `This` position of `new` expressions must always be $\star$, so no actual parameter is named at initialisation. New objects are given existential types with the `This` existentially quantified (bounded above by the `Owner` parameter), which ensures that the actual `This` parameter can never be named directly. The extra well-formedness premise in T-New is stricter than the usual well-formedness

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

$$\frac{\Delta \vdash T \text{ OK}}{\Delta; \Gamma \vdash \texttt{null} : T}$$

(T-Null)

$$\frac{\Delta; \Gamma \vdash e : \exists \Delta' . N \qquad fType(\texttt{f}, N) = T' \qquad \Delta; \Gamma \vdash e' : T \qquad \Delta, \Delta' \vdash T <: T'}{\Delta; \Gamma \vdash e.\texttt{f = } e' : T}$$

(T-Assign)

$$\frac{\Delta \vdash \overline{T}, \mathcal{T} \text{ OK} \qquad \Delta \vdash \exists \texttt{O} \rightarrow \texttt{[}\bot\ \mathcal{T}\texttt{]}.\texttt{C<}\overline{T}\texttt{,}\mathcal{T}\texttt{,O>} \text{ OK}}{\Delta; \Gamma \vdash \texttt{new C<}\overline{T}\texttt{,}\mathcal{T}\texttt{,}\star\ \texttt{>} : \exists \texttt{O} \rightarrow \texttt{[}\bot\ \mathcal{T}\texttt{]}.\texttt{C<}\overline{T}\texttt{,}\mathcal{T}\texttt{,O>}}$$

(T-New)

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

$$\frac{\begin{array}{c}\Delta = \overline{\texttt{X}\rightarrow\texttt{[}\bot\ \ T_u\texttt{]}}, \texttt{Owner}\rightarrow\texttt{[}\bot\ \tau_o\texttt{]},\texttt{This}\rightarrow\texttt{[}\bot\ \texttt{Owner]},\overline{\texttt{O}\rightarrow\texttt{[}\bot\ \tau_u\texttt{]}} \\ \emptyset \vdash \Delta \text{ OK} \qquad \Delta \vdash N,\ \overline{T} \text{ OK} \\ \overline{\mathcal{X}} = \overline{\texttt{X}}, \overline{\texttt{O}}, \texttt{Owner}, \texttt{This} \qquad \Delta; \texttt{this:C<}\overline{\mathcal{X}}\texttt{>} \vdash \overline{M} \text{ OK in C} \\ N = \texttt{D<}\overline{\mathcal{T}}\texttt{,Owner,This>} \qquad \Delta \vdash N <: \texttt{Object<Owner,This>}\end{array}}{\vdash \texttt{class C<}\overline{\texttt{X}\lhd\ T_u}\texttt{,}\ \overline{\texttt{O}\lhd\ \tau_u}\texttt{,Owner}\lhd\ \tau_o\texttt{,This}\lhd\ \texttt{Owner}\ \texttt{>} \lhd\ N\{\overline{\texttt{T f;}}\ \ \overline{M}\} \text{ OK}}$$

(T-Class)

Fig. 3. Selected Tame FJ$_{Own}$ expression and class typing rules.

premise and ensures that the type parameters are well-formed without the extra, quantified parameter in the environment.

We add a standard rule for casting (T-Cast). Unlike in Featherweight Java, we do not distinguish between up-, down-, and stupid casts: we only support downcasts. Although casting is safe in our formal system (because type parameters are preserved at runtime), our implementation is in Java which uses erasure semantics, therefore casting is strictly unsound (as in Java).

In T-Class we enforce that the declared upper bound of `This` is `Owner`. The last two premises ensure that declared classes fall under the '`Object` hierarchy' and are not subtypes of `World`, which means they cannot be used as context parameters, and that the `Owner` and `This` parameters are invariant with respect to inheritance. The latter is an important sanity condition of our encoding of ownership and corresponds to the well-known condition of inheritance and ownership [15]. We assume that `Object` is declared with parameters `Owner` and `This` with the usual bounds.

Operational Semantics The most interesting change from Tame FJ is in object creation:

$$\frac{\iota \notin dom(\mathcal{H}) \qquad fields(\texttt{C}) = \overline{\texttt{f}} \qquad \mathcal{H}' = \mathcal{H}, \iota \rightarrow \{\texttt{C<}\overline{\mathcal{T}}\texttt{,}\mathcal{T}\texttt{,}\iota\texttt{>};\overline{\texttt{f}\rightarrow\texttt{null}}\}}{\texttt{new C<}\overline{\mathcal{T}}\texttt{,}\mathcal{T}\texttt{,}\star\ \texttt{>};\mathcal{H} \rightsquigarrow \iota;\mathcal{H}'}$$

(R-New)

A new object's runtime type (stored in the heap) is formed by replacing the $\star$ used in the program source by the new object's address. Together with the usual rules of substitution (in method invocation), occurances of both `this` and `This` in class declarations are replaced by the instantiation's address, unifying the two representations of the object. Together with the quantification in T-New, objects are, in effect, packed into existential types, with the object's address as witness 'type'. Other operational semantics rules are given in the accompanying technical report [1].

4.1 Discussion

Ownership types are intrinsically dependent because they reflect objects' positions in the heap. We have shown that ownership types can be encoded as parametric types in a Java-like type system, reminiscent of *phantom types* [21]. Phantom types are parametric types where the type parameters are not used as types[4]. Phantom types are used in Haskell to simulate values in types, without the complexity and decidability issues of full dependent types [21]. This is exactly what our system is doing with respect to ownership information. We conclude then, that ownership types systems are, in some sense, no more complex than standard parametric type systems such as Java's. Despite their dependent character, the full power of dependent types is not required to support ownership types systems. We should not overstep the mark, however, and assume that type parametricity is the only, or even the best, foundational model for ownership types.

Most of the ownership features described in Sect. 3 can be accommodated in Tame FJ_{Own}. Inner classes require encoding and are discussed below. Paths of final fields cannot easily be encoded in our formal system. Generic Universe Types [19] can be accommodated after encoding. Ownership domains would require a small extension to the formal system, which we have avoided for the sake of simplicity: each class has a list of `This` type parameters rather than a single parameter. Each parameter represents a domain. Since this change merely changes `This` to $\overline{\texttt{This}}$, we expect very few changes to be necessary to accommodate it.

The extensions to support ownership domains and inner classes (below) are fairly superficial changes, modifying only the restrictions on type parameters and which type parameters are hidden in T-New.

Inner Classes Supporting inner classes would require an extension to Tame FJ_{Own}. References to the surrounding object and the type parameters of the surrounding object must be made available to objects of the inner class. Extending Tame FJ_{Own} could be done by adopting a nesting of classes and objects in the class table and heap or by adding a field to each class pointing to the surrounding object and type parameters for the surrounding classes' type parameters; object creation becomes more complex, but otherwise the calculus is not changed too

[4] More precisely, phantom type parameters are not used on the right hand side of the definition of a type constructor.

much. Classes are essentially encoded, the iterator as inner class example from Sect. 3.1 is encoded as (we elide bounds):

```
class Iterator<D, L_Owner, L_This, It_Owner, It_This> {
    List<D, L_Owner, L_This> out;
    Node<D, L_This, ?> curNode;
    Object<It_This, ?> privField;

    Object<D, ?> next() {...}
}

class Client<Owner, This> {
    <LT> void m(List<World, This, LT> l) {
        Iterator<World, This, LT, This, ?> i
            = new Iterator<World, This, LT, This, ?>();
        i.out = l;
        Object<World, ?> first = i.next();
    }
}
```

Java

We must use (a presumably capture converted) type variable (`LT`) for the `This` parameter of `l`, provide `l`'s type parameters to `i`, and must instantiate the `out` field of `i`.

Type Soundness We have proved type soundness for Tame FJ$_{Own}$ in the usual way [29] by proving progress and preservation theorems. For the most part, our proofs follow those of Tame FJ [11]; they can be downloaded from [1].

In standard existential type systems, witness types are known at runtime, and type soundness guarantees that no type errors involving witness types occur, even though the type system has only partial knowledge of these types during type checking. Taking this approach with Tame FJ$_{Own}$ would not be very informative, since all witness types (according to T-New) will be $\star$. Our static types hold *more* information (the ownership hierarchy) than is represented by the 'witness types'. Our soundness result proves that Tame FJ$_{Own}$ does enforce the ownership hierarchy, i.e., Tame FJ$_{Own}$ enforces not only strict type soundness (well-typed programs won't access non-existent fields or methods), but also that objects reside in the context desribed by their type. Ownership information is represented at runtime by storing the object's address into its `This` position (in R-New): the address propagates into other ownership positions by substitution (in R-Invk).

In proving type soundness for Tame FJ$_{Own}$, we have proved that a one-stage type checker (corresponding to an integration of our pre-processor and the Java type checker) is sound, rather than proving that a two-stage type checker (corresponding to pre-processing and then Java type checking, as in our implementation) is sound. Our approach is theoretically more direct and reflects what we envision to be the long term use of our techniques.

5 Implementation

We have implemented compilers for Java with ownership types and Generic Universe Types by using the techniques described in this paper. Our implementations are simple source to source translators which translate source code to plain Java; the Java compiler is then used to type check and compile the code. Most type errors are caught by the Java compiler, only a few are handled by our translators. Our translators are extensions to the parser and AST elements of the JKit Java compiler [25]. We encode one class at a time and do not need to be aware of the whole program. Generated classes will behave well together, but will be incompatible with plain Java classes[5].

Our approach supports ownership and universe types on top of nearly the entire Java Language, including generics, arrays (including the various kinds of array initialisers; we only support ownership information on the elements in an array, the array itself (like primitive types) is not considered to have an owner), interfaces, inner classes (but not anonymous classes), statics, and wildcards.

Our implementations are very much prototypes, an industrial strength compiler would integrate the encoding with Java type checking, as opposed to our two-stage process. Integration would allow for meaningful error messages and support for effects and encapsulation properties. Furthermore, to be usable, a language requires more than a compiler, libraries must be supported, either by support for non-ownership aware classes (currently, all classes must be written with ownership types) or by producing a set of ownership annotated libraries (or a combination of the two approaches). Our compilers can be downloaded [1].

5.1 Ownership Types

Our source syntax is mostly similar to that used throughout this paper. We support owners, context parameters, orthogonal generics, context- and type-parametric methods, final method parameters as contexts (for dynamic aliases), existential quantification in the form of context wildcards, and inner classes with access to the contexts and context parameters of the surrounding object. We do not support local variables (other than method parameters) or fields as contexts. We support standard casting, including to wildcard owners, but do not directly support "existential owners". Since Java implements generics using erasure [22], the runtime checks on casts do not ensure correctness of ownership parameters; with respect to ownership types, casting merely ensures that a program can type check at compile time. This is not a problem in our formalism, because we do not erase type parameters in the operational semantics.

The obvious function of our compiler is to strip owners and context parameters and replace them with type parameters, this is done both in class declarations and in types; in the latter case, using wildcards in the `This` position.

[5] Strictly, since we generate plain Java, one could write classes which behave well with the generated classes, but not in a way which behaves nicely with the source classes.

Java does not allow wildcard parameters when objects are instantiated: to get around this we use the `Owner` type parameter in the `This` position (because it is the only type parameter which satisfies the declared bound) and immediately cast to the required wildcard type (which inherits the upper bound),

```
new world:Object()                                          //source syntax
new Object<World, ?>()                                       //pseudo-Java
(OwnedObject<World, ?>) new OwnedObject<World, World>()   //Java
```

Note also that, as in OGJ, we have to add an `OwnedObject` which extends `Object` at the root of our class hierarchy to take the encoded ownership parameters. All classes must extend this class (rather than `Object`, which may happen implicitly) and all uses of `Object` changed to `OwnedObject`. In the source syntax, the object's owner is implicit in the extends clause, and so translation of the superclass type must be treated differently from other types. Because we add `OwnedObject` and `World` to our runtime, we must import these classes into each encoded class file.

5.2 Generic Universe Types

The source syntax is pretty standard for generic universes, e.g., `rep List<any Object>`. The translation is much simpler than for ownership types since we do not have to translate context parameters, only types. Most of the issues faced are similar, and simpler, than in the ownership types case: we must check for universe modifiers on all types (but not in `extends` clauses, because ownership is invariant with respect to inheritance, that is superclasses must be peers), `Object` is translated to `OwnedObject`, and care must be taken with array types. As with our ownership types implementation and Java, casting is unsafe and allows for Universe modifiers to be changed improperly; Universes usually requires safe down-casting.

6 Conclusion and Future Work

In this paper we have shown how ownership types, Generic Universe Types, Ownership Domains, and a range of extensions to ownership types systems can be encoded using Java Generics and wildcards. The key concepts are the representation of context parameters as type parameters, the reification of `this` as a type parameter, the hiding of `this` using wildcards, and the phantom ownership hierarchy thus created. Our developments shed light on the type-theoretic foundations of ownership types and offer a route for practical compilers constructed upon existing technology.

Future Work The main thrust of future work will be in supporting owners-as-dominators, and other encapsulation polices and effects, in our formal work and compilers. This would require integrating our translating compiler with an

existing Java compiler, which would also allow for better error messages and more efficient type checking. We would also like to encode libraries with ownership type information for use with our compilers. An alternative would be to develop our encoding so that encoded classes and the un-annotated Java libraries can interact together.

References

1. Accompanying webpage. https://ecs.victoria.ac.nz/Main/Encoding.
2. Marwan Abi-Antoun and Jonathan Aldrich. Ownership Domains in the Real World. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, 2008.
3. Jonathan Aldrich and Craig Chambers. Ownership Domains: Separating Aliasing Policy from Mechanism. In *European Conference on Object Oriented Programming (ECOOP)*, 2004.
4. 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.
5. Robert L. Bocchino, Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A Type and Effect System for Deterministic Parallel Java. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2009.
6. Chandrasekhar Boyapati, Robert Lee, and Martin C. Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2002.
7. Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership Types for Object Encapsulation. In *Principles of Programming Languages (POPL)*, 2003.
8. Nicholas Cameron. *Existential Types for Variance — Java Wildcards and Ownership Types*. PhD thesis, Imperial College London, 2009.
9. Nicholas Cameron and Werner Dietl. Comparing Universes and Existential Ownership Types. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, 2009.
10. Nicholas Cameron and Sophia Drossopoulou. Existential Quantification for Variant Ownership. In *European Symposium on Programming Languages and Systems (ESOP)*, 2009.
11. Nicholas Cameron, Sophia Drossopoulou, and Erik Ernst. A Model for Java with Wildcards. In *European Conference on Object Oriented Programming (ECOOP)*, 2008.
12. Nicholas Cameron, Sophia Drossopoulou, James Noble, and Matthew Smith. Multiple Ownership. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2007.
13. Nicholas Cameron and James Noble. OGJ Gone Wild. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, 2009.
14. David G. Clarke. *Object Ownership and Containment*. PhD thesis, School of Computer Science and Engineering, The University of New South Wales, Sydney, Australia, 2001.

15. David G. Clarke and Sophia Drossopoulou. Ownership, Encapsulation and the Disjointness of Type and Effect. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2002.
16. David G. Clarke, John M. Potter, and James Noble. Ownership Types for Flexible Alias Protection. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1998.
17. David Cunningham, Werner Dietl, Sophia Drossopoulou, Adrian Francalanza, Peter Müller, and Alexander J. Summers. Universe Types for Topology and Encapsulation. In *Formal Methods for Components and Objects (FMCO)*, 2008.
18. David Cunningham, Sophia Drossopoulou, and Susan Eisenbach. Universe Types for Race Safety. In *Verification and Analysis of Multi-threaded Java-like Programs (VAMP)*, 2007.
19. Werner Dietl, Sophia Drossopoulou, and Peter Müller. Generic Universe Types. In *European Conference on Object Oriented Programming (ECOOP)*, 2007.
20. James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Third Edition*. Addison-Wesley, Boston, Mass., 2005.
21. Ralf Hinze. *The Fun of Programming*, pages 245–262. Palgrave Macmillan, 2003. Fun with phantom types.
22. 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.
23. Yi Lu and John Potter. On Ownership and Accessibility. In *European Conference on Object Oriented Programming (ECOOP)*, 2006.
24. Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular Invariants for Layered Object Structures. *Science of Computer Programming*, 62(3):253–286, October 2006.
25. David Pearce. Jkit compiler. http://www.ecs.vuw.ac.nz/~djp/jkit.
26. Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. Featherweight Generic Confinement. *J. Funct. Program.*, 16(6):793–811, 2006.
27. Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. Generic Ownership for Generic Java. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2006.
28. Mads Torgersen, Christian Plesner Hansen, Erik Ernst, Peter von der Ahé, Gilad Bracha, and Neal Gafter. Adding Wildcards to the Java Programming Language. *Journal of Object Technology*, 3(11):97–116, 2004. Special issue: OOPS track at SAC 2004, Nicosia/Cyprus.
29. Andrew K. Wright and Matthias Felleisen. A Syntactic Approach to Type Soundness. *Information and Computation*, 115(1):38–94, 1994.
30. Tobias Wrigstad. *Ownership-Based Alias Managemant*. PhD thesis, KTH, Sweden, 2006.
31. Tobias Wrigstad and David G. Clarke. Existential Owners for Ownership Types. *Journal of Object Technology*, 6(4), 2007.