Generic Universe Types

Werner Dietl[1], Sophia Drossopoulou[2], and Peter Müller[1]

[1] ETH Zurich
{Werner.Dietl,Peter.Mueller}@inf.ethz.ch
[2] Imperial College London
S.Drossopoulou@imperial.ac.uk

Abstract. Ownership is a powerful concept to structure the object store and to control aliasing and modifications of objects. This paper presents an ownership type system for a Java-like programming language with generic types. Like our earlier Universe type system, Generic Universe Types enforce the owner-as-modifier discipline. This discipline does not restrict aliasing, but requires modifications of an object to be initiated by its owner. This allows owner objects to control state changes of owned objects, for instance, to maintain invariants. Generic Universe Types require a small annotation overhead and provide strong static guarantees. They are the first type system that combines the owner-as-modifier discipline with type genericity.

1 Introduction

The concept of object ownership allows programmers to structure the object store hierarchically and to control aliasing and access between objects. Ownership has been applied successfully to various problems, for instance, program verification [18,20,21], thread synchronization [5,15], memory management [2,6], and representation independence [3].

Existing ownership models share fundamental concepts: Each object has at most one owner object. The set of all objects with the same owner is called a *context*. The *root context* is the set of objects with no owner. The ownership relation is a tree order.

However, existing models differ in the restrictions they enforce. The original ownership types [9] and their descendants [4,7,8,24] restrict aliasing and enforce the *owner-as-dominator* discipline: All reference chains from an object in the root context to an object `o` in a different context go through `o`'s owner. This severe restriction of aliasing is necessary for some of the applications of ownership, for instance, memory management and representation independence.

However, for applications such as program verification, restricting aliasing is not necessary. Instead, it suffices to enforce the *owner-as-modifier* discipline: An object `o` may be referenced by any other object, but reference chains that do not pass through `o`'s owner must not be used to modify `o`. This allows owner objects to control state changes of owned objects and thus maintain invariants. The owner-as-modifier discipline has been inspired by Flexible Alias Protection [23]. It is enforced

E. Ernst (Ed.): ECOOP 2007, LNCS 4609, pp. 28–53, 2007.

by the Universe type system [12], in Spec#'s dynamic ownership model [18], and Effective Ownership Types [19]. The owner-as-modifier discipline imposes weaker restrictions than the owner-as-dominator discipline, which allows it to handle common implementations where objects are shared between objects, such as collections with iterators, shared buffers, or the Flyweight pattern [12,22]. Some implementations can be slightly adapted to satisfy the owner-as-modifier discipline, for example an iterator can delegate modifications to the corresponding collection which owns the internal representation.

Although ownership type systems have covered all features of Java-like languages (including for example exceptions, inner classes, and static class members) there are only three proposals of ownership type systems that support generic types. SafeJava [4] supports type parameters and ownership parameters independently, but does not integrate both forms of parametricity. This leads to significant annotation overhead. Ownership Domains [1] combine type parameters and domain parameters into a single parameter space and thereby reduce the annotation overhead. However, their formalization does not cover type parameters. Ownership Generic Java (OGJ) [24] allows programmers to attach ownership information through type parameters. For instance, a collection of `Book` objects can be typed as "my collection of library books", expressing that the collection object belongs to the current `this` object, whereas the `Book` objects in the collection belong to an object "library". OGJ enforces the owner-as-dominator discipline. It piggybacks ownership information on type parameters. In particular, each class `C` has a type parameter to encode the owner of a `C` object. This encoding allows OGJ to use a slight adaptation of the normal Java type rules to also check ownership, which makes the formalization very elegant.

However, OGJ cannot be easily adapted to enforce the owner-as-modifier discipline. For example, OGJ would forbid a reference from the iterator (object 6) in Fig. 1 to a node (object 5) of the map (object 3), because the reference bypasses the node's owner. However, such references are necessary, and are legal in the owner-as-modifier discipline. A type system can permit such references in two ways.

First, if the iterator contained a field `theMap` that references the associated map object, then path-dependent types [1,4] can express that the `current` field of the iterator points to a `Node` object that is owned by `theMap`. Unfortunately, path-dependent types require the fields on the path (here, `theMap`) to be final, which is too restrictive for many applications.

Second, one can loosen up the static ownership information by allowing certain references to point to objects in any context [12]. Subtyping allows values with specific ownership information to be assigned to "any" variables, and downcasts with runtime checks can be used to recover specific ownership information from such variables. In OGJ, this subtype relation between any-types and other types would require covariant subtyping, for instance, that `Node<This>` is a subtype of `Node<Any>`, which is not supported in Java (or C#). Therefore, piggybacking ownership on the standard Java type system is not possible in the presence of "any".

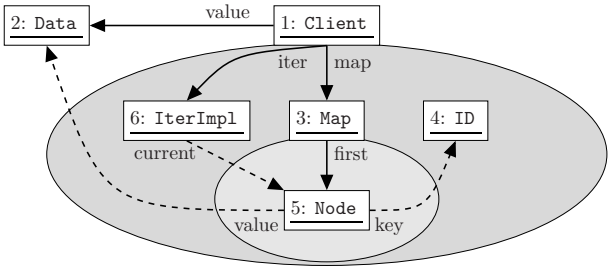

Fig. 1. Object structure of a map from `ID` to `Data` objects. The map is represented by `Node` objects. The iterator has a direct reference to a node. Objects, references, and contexts are depicted by rectangles, arrows, and ellipses, respectively. Owner objects sit atop the context of objects they own. Arrows are labeled with the name of the variable that stores the reference. Dashed arrows depict references that cross context boundaries without going through the owner. Such references must not be used to modify the state of the referenced objects.

In this paper, we present Generic Universe Types (GUT), an ownership type system for a programming language with generic types similar to Java 5 and C# 2.0. GUT enforces the owner-as-modifier discipline using an `any` ownership modifier (analogous to the `readonly` modifier in non-generic Universe types [12]). Our type system supports type parameters for classes and methods. The annotation overhead for programmers is as low as in OGJ, although the presence of `any` makes the type rules more involved. A particularly interesting aspect of our work is how generics and ownership can be combined in the presence of an `any` modifier, in particular, how a restricted form of ownership covariance can be permitted without runtime checks.

Outline. Sec. 2 illustrates the main concepts of GUT by an example. Secs. 3 and 4 present the type rules and the runtime model of GUT, respectively. Sec. 5 presents the type safety and the owner-as-modifier property theorems. Details and proofs can be found in the accompanying technical report [10].

2 Main Concepts

In this section, we explain the main concepts of GUT informally by an example. Class `Map` (Fig. 2) implements a generic map from keys to values. Key-value pairs are stored in singly-linked `Node` objects. Class `Node` extends the superclass `MapNode` (both Fig. 3), which is used by the iterator (classes `Iter` and `IterImpl` in Fig. 4). The `main` method of class `Client` (Fig. 5) builds up the map structure shown in Fig. 1. For simplicity, we omit access modifiers from all examples.

Ownership Modifiers. A type in GUT is either a type variable or consists of an ownership modifier, a class name, and possibly type arguments. The *ownership modifier* expresses object ownership relative to the current receiver object

`this`[1]. Programs may contain the ownership modifiers `peer`, `rep`, and `any`. `peer` expresses that an object has the same owner as the `this` object, `rep` expresses that an object is owned by `this`, and `any` expresses that an object may have any owner. `any` types are supertypes of the `rep` and `peer` types with the same class and type arguments because they convey less specific ownership information.

The use of ownership modifiers is illustrated by class `Map` (Fig. 2). A `Map` object owns its `Node` objects since they form the internal representation of the map and should, therefore, be protected from unwanted modifications. This ownership relation is expressed by the `rep` modifier of `Map`'s field `first`, which points to the first node of the map.

```
class Map<K, V> {
    rep Node<K, V> first;

    void put(K key, V value) {
        rep Node<K, V> newfirst = new rep Node<K, V>();
        newfirst.init(key, value, first);
        first = newfirst;
    }

    pure V get(K key) {
        peer Iter<K, V> i = iterator();
        while (i.hasNext()) {
            if (i.getKey().equals(key)) return i.getValue();
            i.next();
        }
        return null;
    }

    pure peer Iter<K, V> iterator() {
        peer IterImpl<K, V, rep Node<K, V> > res;
        res = new peer IterImpl<K, V, rep Node<K, V> >();
        res.setCurrent(first);
        return res;
    }

    pure peer IterImpl<K, V, rep Node<K, V> > altIterator() {
      /* same implementation as method iterator() above */
    }
}
```

Fig. 2. An implementation of a generic map. `Map` objects own their `Node` objects, as indicated by the `rep` modifier in all occurrences of class `Node`. Method `altIterator` is for illustration purposes only.

The owner-as-modifier discipline is enforced by disallowing modifications of objects through `any` references. That is, an expression of an `any` type may be

[1] We ignore static methods in this paper, but an extension is possible [20].

used as receiver of field reads and calls to side-effect free (*pure*) methods, but not of field updates or calls to non-pure methods. To check this property, we require side-effect free methods to be annotated with the keyword `pure`.

Viewpoint Adaptation. Since ownership modifiers express ownership relative to `this`, they have to be adapted when this "viewpoint" changes. Consider `Node`'s inherited method `init` (Fig. 3). After substituting the type variable `X`, the third parameter has type `peer Node<K,V>`. The `peer` modifier expresses that the parameter object must have the same owner as the receiver of the method. On the other hand, `Map`'s method `put` calls `init` on a `rep Node` receiver, that is, an object that is owned by `this`. Therefore, the third parameter of the call to `init` also has to be owned by `this`. This means that from this particular call's viewpoint, the third parameter needs a `rep` modifier, although it is declared with a `peer` modifier. In the type system, this *viewpoint adaptation* is done by combining the type of the receiver of a call (here, `rep Node<K,V>`) with the type of the formal parameter (here, `peer Node<K,V>`). This combination yields the argument type from the caller's point of view (here, `rep Node<K,V>`).

```
class MapNode<K, V, X extends peer MapNode<K, V, X> > {
    K key; V value; X next;

    void init(K k, V v, X n) { key = k; value = v; next = n; }
}

class Node<K, V> extends MapNode<K, V, peer Node<K, V> > {}
```

Fig. 3. Nodes form the internal representation of maps. Class `MapNode` implements nodes for singly-linked lists. Using a type variable for the type of `next` is useful to implement iterators. The subclass `Node` instantiates `MapNode`'s type parameter `X` to implement a list of nodes with the same owner.

Viewpoint adaptation and the owner-as-modifier discipline provide encapsulation of internal representation objects. Assume that class `Map` by mistake leaked a reference to an internal node, for instance, by making `first` public or by providing a method that returns the node. By viewpoint adaptation of the node type, `rep Node<K,V>`, clients of the map can only obtain an `any` reference to the node and, thus, the owner-as-modifier discipline guarantees that clients cannot directly modify the node structure. This allows the map to maintain invariants over the node, for instance, that the node structure is acyclic.

Type Parameters. Ownership modifiers are also used in actual type arguments. For instance, `Map`'s method `iterator` instantiates `IterImpl` with the type arguments `K`, `V`, and `rep Node<K,V>`. Thus, local variable `res` has type `peer IterImpl<K,V,rep Node<K,V>>`, which has two ownership modifiers. The *main modifier* `peer` expresses that the `IterImpl` object has the same owner as `this`, whereas the *argument modifier* `rep` expresses that the `Node` objects used

by the iterator are owned by `this`. It is important to understand that this argument modifier again expresses ownership relative to the current `this` object (here, the `Map` object), and not relative to the instance of the generic class that contains the argument modifier (here, the `IterImpl` object `res`).

```
interface Iter<K, V> {
    pure K getKey();
    pure V getValue();
    pure boolean hasNext();
    void next();
}

class IterImpl<K, V, X extends any MapNode<K, V, X>>
implements Iter<K, V> {
    X current;

    void setCurrent(X c) { current = c; }
    pure K getKey() { return current.key; }
    pure V getValue() { return current.value; }
    pure boolean hasNext() { return current != null; }
    void next() { current = current.next; }
}
```

Fig. 4. Class `IterImpl` implements iterators over `MapNode` structures. The precise node type is passed as type parameter. The upper bound allows methods to access a node's fields. Interface `Iter` hides `IterImpl`'s third type parameter from clients.

Type variables have upper bounds, which default to `any Object`. In a class `C`, the ownership modifiers of an upper bound express ownership relative to the `C` instance `this`. However, when `C`'s type variables are instantiated, the modifiers of the actual type arguments are relative to the receiver of the method that contains the instantiation. Therefore, checking the conformance of a type argument to its upper bound requires a viewpoint adaptation. For instance, to check the instantiation `peer IterImpl<K,V,rep Node<K,V>>` in class `Map`, we adapt the upper bound of `IterImpl`'s type variable `X` (`any MapNode<K,V,X>`) from viewpoint `peer IterImpl<K,V,rep Node<K,V>>` to the viewpoint `this`. With the appropriate substitutions, this adaptation yields `any MapNode<K,V,rep Node<K,V>>`. The actual type argument `rep Node<K,V>` is a subtype of the adapted upper bound. Therefore, the instantiation is correct. The `rep` modifier in the type argument and the adapted upper bound reflects correctly that the `current` node of this particular iterator is owned by `this`.

Type variables are not subject to the viewpoint adaptation that is performed for non-variable types. When type variables are used, for instance, in field declarations, the ownership information they carry stays implicit and does, therefore, not have to be adapted. The substitution of type variables by their actual type arguments happens in the scope in which the type variables were instantiated. Therefore, the

viewpoint is the same as for the instantiation, and no viewpoint adaptation is required. For instance, the call expression `iter.getKey()` in method `main` (Fig. 5) has type `rep ID`, because the result type of `getKey()` is the type variable `K`, which gets substituted by the first type argument of `iter`'s type, `rep ID`.

Thus, even though an `IterImpl` object does not know the owner of the keys and values (due to the implicit `any` upper bound for `K` and `V`), clients of the iterator can recover the exact ownership information from the type arguments. This illustrates that Generic Universe Types provide strong static guarantees similar to those of owner-parametric systems [9], even in the presence of `any` types. The corresponding implementation in non-generic Universe types requires a downcast from the `any` type to a `rep` type and the corresponding runtime check [12].

```
class ID { /* ... */ }
class Data { /* ... */ }

class Client {
    void main(any Data value) {
        rep Map<rep ID, any Data> map = new rep Map<rep ID, any Data>();
        map.put(new rep ID(), value);

        rep Iter<rep ID, any Data> iter = map.iterator();
        rep ID id = iter.getKey();
    }
}
```

Fig. 5. Main program for our example. The execution of method `main` creates the object structure in Fig. 1.

Limited Covariance and Viewpoint Adaptation of Type Arguments. Subtyping with covariant type arguments is in general not statically type safe. For instance, if `List<String>` were a subtype of `List<Object>`, then clients that view a string list through type `List<Object>` could store `Object` instances in the string list, which breaks type safety. The same problem occurs for the ownership information encoded in types. If `peer IterImpl<K,V,rep Node<K,V>>` were a subtype of `peer IterImpl<K,V,any Node<K,V>>`, then clients that view the iterator through the latter type could use method `setCurrent` (Fig. 4) to set the iterator to a `Node` object with an arbitrary owner, even though the iterator requires a specific owner. The covariance problem can be prevented by disallowing covariant type arguments (like in Java and C#), by runtime checks, or by elaborate syntactic support [13].

However, the owner-as-modifier discipline supports a limited form of covariance without any additional checks. Covariance is permitted if the main modifier of the supertype is `any`. For example, `peer IterImpl<K,V,rep Node<K,V>>` is an admissible subtype of `any IterImpl<K,V,any Node<K,V>>`. This is safe because the owner-as-modifier discipline prevents mutations of objects referenced

through `any` references. In particular, it is not possible to set the iterator to an `any Node` object, which prevents the unsoundness illustrated above.

Besides subtyping, GUT provides another way to view objects through different types, namely viewpoint adaptation. If the adaptation of a type argument yields an `any` type, the same unsoundness as through covariance could occur. Therefore, when a viewpoint adaptation changes an ownership modifier of a type argument to `any`, it also changes the main modifier to `any`.

This behavior is illustrated by method `main` of class `Client` in Fig. 5. Assume that `main` calls `altIterator()` instead of `iterator()`. As illustrated by Fig. 1, the most precise type for the call expression `map.altIterator()` would be `rep IterImpl<rep ID, any Data, any Node<rep ID, any Data>>` because the `IterImpl` object is owned by the `Client` object `this` (hence, the main modifier `rep`), but the nodes referenced by the iterator are neither owned by `this` nor peers of `this` (hence, `any Node`). However, this viewpoint adaptation would change an argument modifier of `altIterator`'s result type from `rep` to `any`. This would allow method `main` to use method `setCurrent` to set the iterator to an `any Node` object and is, thus, not type safe. The correct viewpoint adaptation yields `any IterImpl<rep ID, any Data, any Node<rep ID, any Data>>`. This type is safe, because it prevents the `main` method from mutating the iterator, in particular, from calling the non-pure method `setCurrent`.

Since `next` is also non-pure, `main` must not call `iter.next()` either, which renders `IterImpl` objects useless outside the associated `Map` object. To solve this issue, we provide interface `Iter`, which does not expose the type of internal nodes to clients. The call `map.iterator()` has type `rep Iter<rep ID, any Data>`, which does allow `main` to call `iter.next()`. Nevertheless, the type variable `X` for the type of `current` in class `IterImpl` is useful to improve static type safety. Since the current node is neither a `rep` nor a `peer` of the iterator, the only alternative to a type variable is an `any` type. However, an `any` type would not capture the relationship between an iterator and the associated list. In particular, it would allow clients to use `setCurrent` to set the iterator to a node of an arbitrary map. For a discussion of alternative designs see [10].

3 Static Checking

In this section, we formalize the compile time aspects of GUT. We define the syntax of the programming language, formalize viewpoint adaptation, define subtyping and well-formedness conditions, and present the type rules.

3.1 Programming Language

We formalize Generic Universe Types for a sequential subset of Java 5 and C# 2.0 including classes and inheritance, instance fields, dynamically-bound methods, and the usual operations on objects (allocation, field read, field update, casts). For simplicity, we omit several features of Java and C# such as interfaces, exceptions, constructors, static fields and methods, inner classes, primitive types

and the corresponding expressions, and all statements for control flow. We do not expect that any of these features is difficult to handle (see for instance [4,11,20]). The language we use is similar to Featherweight Generic Java [14]. We added field updates because the treatment of side effects is essential for ownership type systems and especially the owner-as-modifier discipline.

Fig. 6 summarizes the syntax of our language and our naming conventions for variables. We assume that all identifiers of a program are globally unique except for `this` as well as method and parameter names of overridden methods. This can be achieved easily by preceding each identifier with the class or method name of its declaration (but we omit this prefix in our examples).

The superscript $^{\mathtt{s}}$ distinguishes the sorts for static checking from corresponding sorts used to describe the runtime behavior, but is omitted whenever the context determines whether we refer to static or dynamic entities.

$\overline{\mathtt{T}}$ denotes a sequence of `T`s. In such a sequence, we denote the i-th element by $\mathtt{T}_i$. We sometimes use sequences of tuples $S = \overline{\mathtt{X}\ \mathtt{T}}$ as maps and use a function-like notation to access an element $S(\mathtt{X}_i) = \mathtt{T}_i$. A sequence $\overline{\mathtt{T}}$ can be empty. The empty sequence is denoted by ϵ.

A program (`P` $\in$ `Program`) consists of a sequence of classes, the identifier of a main class (`C` $\in$ `ClassId`), and a main expression (`e` $\in$ `Expr`). A program is executed by creating an instance `o` of `C` and then evaluating `e` with `o` as `this` object. We assume that we always have access to the current program `P`, and keep `P` implicit in the notations. Each class (`Cls` $\in$ `Class`) has a class identifier, type variables with upper bounds, a superclass with type arguments, a list of field declarations, and a list of method declarations. `FieldId` is the sort of field identifiers `f`. Like in Java, each class directly or transitively extends the predefined class `Object`.

A type ($^{\mathtt{s}}$`T` $\in$ $^{\mathtt{s}}$`Type`) is either a non-variable type or a type variable identifier (`X` $\in$ `TVarId`). A non-variable type ($^{\mathtt{s}}$`N` $\in$ $^{\mathtt{s}}$`NType`) consists of an ownership modifier, a class identifier, and a sequence of type arguments.

An ownership modifier (`u` $\in$ `OM`) can be $\mathtt{peer}_u$, $\mathtt{rep}_u$, or $\mathtt{any}_u$, as well as the modifier $\mathtt{this}_u$, which is used solely as main modifier for the type of `this`. The modifier $\mathtt{this}_u$ may not appear in programs, but is used by the type system to distinguish accesses through `this` from other accesses. We omit the subscript u if it is clear from context that we mean an ownership modifier.

A method (`mt` $\in$ `Meth`) consists of the method type variables with their upper bounds, the purity annotation, the return type, the method identifier (`m` $\in$ `MethId`), the formal method parameters (`x` $\in$ `ParId`) with their types, and an expression as body. The result of evaluating the expression is returned by the method. `ParId` includes the implicit method parameter `this`.

To be able to enforce the owner-as-modifier discipline, we have to distinguish statically between side-effect free (pure) methods and methods that potentially have side effects. Pure methods are marked by the keyword `pure`. In our syntax, we mark all other methods by `nonpure`, although we omit this keyword in our examples. To focus on the essentials of the type system, we do not include purity checks, but they can be added easily [20].

An expression ($\texttt{e} \in \texttt{Expr}$) can be the `null` literal, method parameter access, field read, field update, method call, object creation, and cast.

Type checking is performed in a type environment (${}^{\texttt{s}}\Gamma \in {}^{\texttt{s}}\texttt{Env}$), which maps the type variables of the enclosing class and method to their upper bounds and method parameters to their types. Since the domains of these mappings are disjoint, we overload the notation, where ${}^{\texttt{s}}\Gamma(\texttt{X})$ refers to the upper bound of type variable `X`, and ${}^{\texttt{s}}\Gamma(\texttt{x})$ refers to the type of method parameter `x`.

$$
\begin{array}{rcl}
\texttt{P} \in \texttt{Program} & ::= & \overline{\texttt{Cls}}\ \texttt{C}\ \texttt{e} \\
\texttt{Cls} \in \texttt{Class} & ::= & \texttt{class C<}\overline{\texttt{X}\ {}^{\texttt{s}}\texttt{N}}\texttt{>}\ \texttt{extends C<}\overline{{}^{\texttt{s}}\texttt{T}}\texttt{>}\ \{\ \overline{\texttt{f}\ {}^{\texttt{s}}\texttt{T}};\ \overline{\texttt{mt}}\ \} \\
{}^{\texttt{s}}\texttt{T} \in {}^{\texttt{s}}\texttt{Type} & ::= & {}^{\texttt{s}}\texttt{N} \mid \texttt{X} \\
{}^{\texttt{s}}\texttt{N} \in {}^{\texttt{s}}\texttt{NType} & ::= & \texttt{u C<}\overline{{}^{\texttt{s}}\texttt{T}}\texttt{>} \\
\texttt{u} \in \texttt{OM} & ::= & \texttt{peer}_u \mid \texttt{rep}_u \mid \texttt{any}_u \mid \texttt{this}_u \\
\texttt{mt} \in \texttt{Meth} & ::= & \texttt{<}\overline{\texttt{X}\ {}^{\texttt{s}}\texttt{N}}\texttt{>}\ \texttt{w}\ {}^{\texttt{s}}\texttt{T}\ \texttt{m(}\overline{\texttt{x}\ {}^{\texttt{s}}\texttt{T}}\texttt{)}\ \{\ \texttt{return e}\ \} \\
\texttt{w} \in \texttt{Purity} & ::= & \texttt{pure} \mid \texttt{nonpure} \\
\texttt{e} \in \texttt{Expr} & ::= & \texttt{null} \mid \texttt{x} \mid \texttt{e.f} \mid \texttt{e.f=e} \mid \texttt{e.m<}\overline{{}^{\texttt{s}}\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{)} \mid \texttt{new}\ {}^{\texttt{s}}\texttt{N} \mid \texttt{(}{}^{\texttt{s}}\texttt{T}\texttt{)}\ \texttt{e} \\
{}^{\texttt{s}}\Gamma \in {}^{\texttt{s}}\texttt{Env} & ::= & \overline{\texttt{X}\ {}^{\texttt{s}}\texttt{N}};\ \overline{\texttt{x}\ {}^{\texttt{s}}\texttt{T}}
\end{array}
$$

Fig. 6. Syntax and type environments

3.2 Viewpoint Adaptation

Since ownership modifiers express ownership relative to an object, they have to be adapted whenever the viewpoint changes. In the type rules, we need to *adapt a type* `T` *from a viewpoint* that is described by another type $\texttt{T}'$ *to the viewpoint* `this`. In the following, we omit the phrase "to the viewpoint `this`". To perform the viewpoint adaptation, we define an overloaded operator $\triangleright$ to: (1) Adapt an ownership modifier from a viewpoint that is described by another ownership modifier; (2) Adapt a type from a viewpoint that is described by an ownership modifier; (3) Adapt a type from a viewpoint that is described by another type.

Adapting an Ownership Modifier w.r.t. an Ownership Modifier. We explain viewpoint adaptation using a field access $\texttt{e}_1\texttt{.f}$. Analogous adaptations occur for method parameters and results as well as upper bounds of type parameters. Let `u` be the main modifier of $\texttt{e}_1$'s type, which expresses ownership relative to `this`. Let $\texttt{u}'$ be the main modifier of `f`'s type, which expresses ownership relative to the object that contains `f`. Then relative to `this`, the type of the field access $\texttt{e}_1\texttt{.f}$ has main modifier $\texttt{u}\triangleright\texttt{u}'$.

$$
\begin{array}{rl}
\triangleright :: \texttt{OM} \times \texttt{OM} \to \texttt{OM} & \\
\texttt{this}\triangleright\texttt{u}' = \texttt{u}' & \texttt{u}\triangleright\texttt{this} = \texttt{u} \\
\texttt{peer}\triangleright\texttt{peer} = \texttt{peer} & \texttt{rep}\triangleright\texttt{peer} = \texttt{rep} \\
\texttt{u}\triangleright\texttt{u}' = \texttt{any} \quad \text{otherwise} &
\end{array}
$$

The field access $\texttt{e}_1\texttt{.f}$ illustrates the motivation for this definition: (1) Accesses through `this` (that is, $\texttt{e}_1$ is the variable `this`) do not require a viewpoint adaptation since the ownership modifier of the field is already relative to `this`.

(2) In the formalization of subtyping (see ST-1) we combine an ownership modifier $\mathtt{u}$ with $\mathtt{this}_u$. Again, this does not require a viewpoint adaptation.

(3) If the main modifiers of both $\mathtt{e}_1$ and $\mathtt{f}$ are `peer`, then the object referenced by $\mathtt{e}_1$ has the same owner as `this` and the object referenced by $\mathtt{e}_1.\mathtt{f}$ has the same owner as $\mathtt{e}_1$ and, thus, the same owner as `this`. Consequently, the main modifier of $\mathtt{e}_1.\mathtt{f}$ is also `peer`. (4) If the main modifier of $\mathtt{e}_1$ is `rep` and the main modifier of $\mathtt{f}$ is `peer`, then the main modifier of $\mathtt{e}_1.\mathtt{f}$ is `rep`, because the object referenced by $\mathtt{e}_1$ is owned by `this` and the object referenced by $\mathtt{e}_1.\mathtt{f}$ has the same owner as $\mathtt{e}_1$, that is, `this`. (5) In all other cases, we cannot determine statically that the object referenced by $\mathtt{e}_1.\mathtt{f}$ has the same owner as `this` or is owned by `this`. Therefore, in these cases the main modifier of $\mathtt{e}_1.\mathtt{f}$ is `any`.

Adapting a Type w.r.t. an Ownership Modifier. As explained in Sec. 2, type variables are not subject to viewpoint adaptation. For non-variable types, we determine the adapted main modifier using the auxiliary function $\triangleright_m$ below and adapt the type arguments recursively:

$$\begin{aligned} \triangleright \;::\; &\mathtt{OM} \times {}^{\mathtt{s}}\mathtt{Type} \to {}^{\mathtt{s}}\mathtt{Type} \\ &\mathtt{u}\triangleright\mathtt{X} = \mathtt{X} \\ &\mathtt{u}\triangleright\mathtt{N} = (\mathtt{u}\triangleright_m\mathtt{N})\ \mathtt{C<}\overline{\mathtt{u}\triangleright\mathtt{T}}\mathtt{>} \qquad \text{where } \mathtt{N} = \mathtt{u}'\ \mathtt{C<}\overline{\mathtt{T}}\mathtt{>} \end{aligned}$$

The adapted main modifier is determined by $\mathtt{u}\triangleright\mathtt{u}'$, except for unsafe (covariance-like) viewpoint adaptations, as described in Sec. 2, in which case it is `any`. Unsafe adaptations occur if at least one of $\mathtt{N}$'s type arguments contains the modifier `rep`, $\mathtt{u}'$ is `peer`, and $\mathtt{u}$ is `rep` or `peer`. This leads to the following definition:

$$\begin{aligned} \triangleright_m \;::\; &\mathtt{OM} \times {}^{\mathtt{s}}\mathtt{NType} \to \mathtt{OM} \\ &\mathtt{u}\triangleright_m\mathtt{u}'\ \mathtt{C<}\overline{\mathtt{T}}\mathtt{>} = \begin{cases} \mathtt{any} & \text{if } (\mathtt{u} = \mathtt{rep} \vee \mathtt{u} = \mathtt{peer}) \wedge \mathtt{u}' = \mathtt{peer} \wedge \mathtt{rep} \in \overline{\mathtt{T}} \\ \mathtt{u}\triangleright\mathtt{u}' & \text{otherwise} \end{cases} \end{aligned}$$

The notation $\mathtt{u} \in \overline{\mathtt{T}}$ expresses that at least one type $\mathtt{T}_i$ or its (transitive) type arguments contain ownership modifier $\mathtt{u}$.

Adapting a Type w.r.t. a Type. We adapt a type $\mathtt{T}$ from the viewpoint described by another type, $\mathtt{u}\ \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}$:

$$\begin{aligned} \triangleright \;::\; &{}^{\mathtt{s}}\mathtt{NType} \times {}^{\mathtt{s}}\mathtt{Type} \to {}^{\mathtt{s}}\mathtt{Type} \\ &\mathtt{u}\ \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}\triangleright\mathtt{T} = (\mathtt{u}\triangleright\mathtt{T})[\overline{\mathtt{T}/\mathtt{X}}] \qquad \text{where } \overline{\mathtt{X}} = \mathrm{dom}(\mathtt{C}) \end{aligned}$$

The operator $\triangleright$ adapts the ownership modifiers of $\mathtt{T}$ and then substitutes the type arguments $\overline{\mathtt{T}}$ for the type variables $\overline{\mathtt{X}}$ of $\mathtt{C}$. This substitution is denoted by $[\overline{\mathtt{T}/\mathtt{X}}]$. Since the type arguments already are relative to `this`, they are not subject to viewpoint adaptation. Therefore, the substitution of type variables happens after the viewpoint adaptation of $\mathtt{T}$'s ownership modifiers. For a declaration $\mathtt{class\ C<}\overline{\mathtt{X}}\ \mathtt{\_>}$ ..., $\mathrm{dom}(\mathtt{C})$ denotes $\mathtt{C}$'s type variables $\overline{\mathtt{X}}$.

Note that the first parameter is a non-variable type, because concrete ownership information $\mathtt{u}$ is needed to adapt the viewpoint and the actual type arguments $\overline{\mathtt{T}}$ are needed to substitute the type variables $\overline{\mathtt{X}}$. In the type rules, subsumption will be used to replace type variables by their upper bounds and thereby obtain a concrete type as first argument of $\triangleright$.

Example. The hypothetical call `map.altIterator()` in `main` (Fig. 5) illustrates the most interesting viewpoint adaptation, which we discussed in Sec. 2. The type of this call is the adaptation of `peer IterImpl<K,V,rep Node<K,V>>` (the return type of `altIterator`) from `rep Map<rep ID,any Data>` (the type of the receiver expression). According to the above definition, we first adapt the return type from the viewpoint of the receiver type, `rep`, and then substitute type variables.

The type arguments of the adapted type are obtained by applying viewpoint adaptation recursively to the type arguments. The type variables `K` and `V` are not affected by the adaptation. For the third type argument, `rep` $\triangleright$ `rep Node<K,V>` yields `any Node<K,V>` because `rep` $\triangleright$ `rep`=`any`, and again because the type variables `K` and `V` are not subject to viewpoint adaptation. Note that here, an ownership modifier of a type argument is promoted from `rep` to `any`. Therefore, to avoid unsafe covariance-like adaptations, the main modifier of the adapted type must be `any`. This is, indeed, the case, as the main modifier is determined by `rep` $\triangleright_m$ `peer IterImpl<K,V,rep Node<K,V>>`, which yields `any`.

So far, the adaptation yields `any IterImpl<K,V,any Node<K,V>>`. Now we substitute the type variables `K` and `V` by the instantiations given in the receiver type, `rep ID` and `any Data`, and obtain the type of the call:

```
any IterImpl<rep ID, any Data, any Node<rep ID,any Data>>
```

3.3 Subclassing and Subtyping

We use the term *subclassing* to refer to the relation on classes as declared in a program by the `extends` keyword, irrespective of main modifiers. *Subtyping* takes main modifiers into account.

Subclassing. The subclass relation $\sqsubseteq$ is defined on instantiated classes, which are denoted by `C<`$\overline{\texttt{T}}$`>`. The subclass relation is the smallest relation satisfying the rules in Fig. 7. Each un-instantiated class is a subclass of the class it extends (SC-1). The form `class C<`$\overline{\texttt{X N}}$`> extends C'<`$\overline{\texttt{T'}}$`> { `$\overline{\texttt{f T}}$`; `$\overline{\texttt{m}}$` }`, or a prefix thereof, expresses that the program contains such a class declaration. Subclassing is reflexive (SC-2) and transitive (SC-3). Subclassing is preserved by substitution of type arguments for type variables (SC-4). Note that such substitutions may lead to ill-formed types, for instance, when the upper bound of a substituted type variable is not respected. We prevent such types by well-formedness rules, presented in Fig. 9.

Subtyping. The subtype relation $<:$ is defined on types. The judgment $\Gamma \vdash \texttt{T} <: \texttt{T}'$ expresses that type `T` is a subtype of type `T'` in type environment Γ. The environment is needed since types may contain type variables. The rules for this subtyping judgment are presented in Fig. 8. Two types with the same main modifier are subtypes if the corresponding classes are subclasses. Ownership modifiers in the `extends` clause ($\overline{\texttt{T'}}$) are relative to the instance of class `C`, whereas the modifiers in a type are relative to `this`. Therefore, $\overline{\texttt{T'}}$ has to be adapted from the viewpoint of the `C` instance to `this` (ST-1). Since both $\texttt{this}_u$ and `peer`

$$\text{SC-1}\ \frac{\texttt{class C<}\overline{\texttt{X}}\texttt{\_> extends C}'\texttt{<}\overline{\texttt{T}'}\texttt{>}}{\texttt{C<}\overline{\texttt{X}}\texttt{>} \sqsubseteq \texttt{C}'\texttt{<}\overline{\texttt{T}'}\texttt{>}} \qquad \text{SC-2}\ \frac{}{\texttt{C<}\overline{\texttt{T}}\texttt{>} \sqsubseteq \texttt{C<}\overline{\texttt{T}}\texttt{>}}$$

$$\text{SC-3}\ \frac{\texttt{C<}\overline{\texttt{T}}\texttt{>} \sqsubseteq \texttt{C}''\texttt{<}\overline{\texttt{T}''}\texttt{>} \quad \texttt{C}''\texttt{<}\overline{\texttt{T}''}\texttt{>} \sqsubseteq \texttt{C}'\texttt{<}\overline{\texttt{T}'}\texttt{>}}{\texttt{C<}\overline{\texttt{T}}\texttt{>} \sqsubseteq \texttt{C}'\texttt{<}\overline{\texttt{T}'}\texttt{>}} \qquad \text{SC-4}\ \frac{\texttt{C<}\overline{\texttt{T}}\texttt{>} \sqsubseteq \texttt{C}'\texttt{<}\overline{\texttt{T}'}\texttt{>}}{\texttt{C<}\overline{\texttt{T}}\texttt{[T}''\texttt{/X}''\texttt{]>} \sqsubseteq \texttt{C}'\texttt{<}\overline{\texttt{T}'}\texttt{[T}''\texttt{/X}''\texttt{]>}}$$

Fig. 7. Rules for subclassing

express that an object has the same owner as `this`, a type with main modifier $\texttt{this}_u$ is a subtype of the corresponding type with main modifier `peer` (ST-2). This rule allows us to treat `this` as an object of a `peer` type. Subtyping is transitive (ST-3). A type variable is a subtype of its upper bound in the type environment (ST-4). Two types are subtypes, if they obey the limited covariance described in Sec. 2 (ST-5). Covariant subtyping is expressed by the relation $<:_{\texttt{a}}$. Covariant subtyping is reflexive (TA-1). A supertype may have more general type arguments than the subtype if the main modifier of the supertype is `any` (TA-2). Note that the sequences $\overline{\texttt{T}}$ and $\overline{\texttt{T}'}$ in rule TA-2 can be empty, which allows one to derive, for instance, `peer Object` $<:_{\texttt{a}}$ `any Object`. Reflexivity of $<:$ follows from TA-1 and ST-5.

$$\text{ST-1}\ \frac{\texttt{C<}\overline{\texttt{T}}\texttt{>} \sqsubseteq \texttt{C}'\texttt{<}\overline{\texttt{T}'}\texttt{>}}{\Gamma \vdash \texttt{u C<}\overline{\texttt{T}}\texttt{>} <: \texttt{u}\triangleright(\texttt{this}_u\ \texttt{C}'\texttt{<}\overline{\texttt{T}'}\texttt{>})} \qquad \text{ST-2}\ \frac{}{\Gamma \vdash \texttt{this}_u\ \texttt{C<}\overline{\texttt{T}}\texttt{>} <: \texttt{peer C<}\overline{\texttt{T}}\texttt{>}}$$

$$\text{ST-3}\ \frac{\Gamma \vdash \texttt{T} <: \texttt{T}'' \quad \Gamma \vdash \texttt{T}'' <: \texttt{T}'}{\Gamma \vdash \texttt{T} <: \texttt{T}'} \qquad \text{ST-4}\ \frac{}{\Gamma \vdash \texttt{X} <: \Gamma(\texttt{X})} \qquad \text{ST-5}\ \frac{\texttt{T} <:_{\texttt{a}} \texttt{T}'}{\Gamma \vdash \texttt{T} <: \texttt{T}'}$$

$$\text{TA-1}\ \frac{}{\texttt{T} <:_{\texttt{a}} \texttt{T}} \qquad \text{TA-2}\ \frac{\overline{\texttt{T} <:_{\texttt{a}} \texttt{T}'}}{\texttt{u C<}\overline{\texttt{T}}\texttt{>} <:_{\texttt{a}} \texttt{any C<}\overline{\texttt{T}'}\texttt{>}}$$

Fig. 8. Rules for subtyping and limited covariance

In our example, using rule TA-1 for `K` and `V`, and rule TA-2 we obtain `rep Node<K,V>` $<:_{\texttt{a}}$ `any Node<K,V>`. Rules TA-2 and ST-5 allow us to derive
`peer IterImpl<K,V,rep Node<K,V>>` $<:$ `any IterImpl<K,V,any Node<K,V>>`,
which is an example for limited covariance. Note that it is not possible to derive
`peer IterImpl<K,V,rep Node<K,V>>` $<:$ `peer IterImpl<K,V,any Node<K,V>>`;
that would be unsafe covariant subtyping as discussed in Sec. 2.

3.4 Lookup Functions

In this subsection, we define the functions to look up the type of a field or the signature of a method.

Field Lookup. The function ${}^{s}fType(\mathtt{C},\mathtt{f})$ yields the type of field `f` as declared in class `C`. The result is undefined if `f` is not declared in `C`. Since identifiers are assumed to be globally unique, there is only one declaration for each field identifier.

$$\text{SFT}\ \frac{\texttt{class C<\_> extends \_<\_> \{ } \ldots \texttt{T f} \ldots \texttt{; \_ \}}}{{}^{s}fType(\mathtt{C},\mathtt{f}) = \mathtt{T}}$$

Method Lookup. The function $mType(\mathtt{C},\mathtt{m})$ yields the signature of method `m` as declared in class `C`. The result is undefined if `m` is not declared in `C`. We do not allow overloading of methods; therefore, the method identifier is sufficient to uniquely identify a method.

$$\text{SMT}\ \frac{\texttt{class C<\_> extends \_<\_> \{\_ ; } \ldots \texttt{<}\overline{\texttt{X}_m\ \texttt{N}_b}\texttt{> w T}_r\ \texttt{m(}\overline{\texttt{x T}_p}\texttt{)} \ldots \texttt{\}}}{mType(\mathtt{C},\mathtt{m}) = \texttt{<}\overline{\texttt{X}_m\ \texttt{N}_b}\texttt{> w T}_r\ \texttt{m(}\overline{\texttt{x T}_p}\texttt{)}}$$

3.5 Well-Formedness

In this subsection, we define well-formedness of types, methods, classes, programs, and type environments. The well-formedness rules are summarized in Fig. 9 and explained in the following.

$$\text{WFT-1}\ \frac{\texttt{X} \in \mathrm{dom}(\Gamma)}{\Gamma \vdash \texttt{X ok}} \qquad \text{WFT-2}\ \frac{\texttt{class C<\_}\ \overline{\texttt{N}}\texttt{>}\ \ldots \quad \Gamma \vdash \overline{\texttt{T}}\ \texttt{ok} \qquad \Gamma \vdash \overline{\texttt{T}} <: ((\texttt{u C<}\overline{\texttt{T}}\texttt{>}) \rhd \overline{\texttt{N}})}{\Gamma \vdash \texttt{u C<}\overline{\texttt{T}}\texttt{> ok}}$$

$$\text{WFM-1}\ \frac{\begin{array}{c}\Gamma = \overline{\texttt{X}_m\ \texttt{N}_b}, \overline{\texttt{X N}}; \texttt{this}\ (\texttt{this}_u\ \texttt{C<}\overline{\texttt{X}}\texttt{>}), \overline{\texttt{x T}_p} \\ \Gamma \vdash \texttt{T}_r, \overline{\texttt{N}_b}, \overline{\texttt{T}_p}\ \texttt{ok} \qquad \Gamma \vdash \texttt{e} : \texttt{T}_r \qquad override(\mathtt{C},\mathtt{m}) \\ \texttt{w} = \texttt{pure} \ \Rightarrow\ (\overline{\texttt{T}_p} = \overline{\texttt{any} \rhd \texttt{T}_p} \ \wedge\ \overline{\texttt{N}_b} = \overline{\texttt{any} \rhd \texttt{N}_b})\end{array}}{\texttt{<}\overline{\texttt{X}_m\ \texttt{N}_b}\texttt{> w T}_r\ \texttt{m(}\overline{\texttt{x T}_p}\texttt{)}\ \{\ \texttt{return e}\ \}\ \texttt{ok in C<}\overline{\texttt{X N}}\texttt{>}}$$

$$\text{WFM-2}\ \frac{\forall\ \texttt{class C'<}\overline{\texttt{X' N'}}\texttt{>}:\quad \texttt{C<}\overline{\texttt{X}}\texttt{>} \sqsubseteq \texttt{C'<}\overline{\texttt{T'}}\texttt{>} \wedge \mathrm{dom}(\mathtt{C}) = \overline{\texttt{X}} \Rightarrow mType(\mathtt{C}',\mathtt{m}) \text{ is undefined } \vee\ mType(\mathtt{C},\mathtt{m}) = mType(\mathtt{C}',\mathtt{m})[\overline{\texttt{T'}/\texttt{X'}}]}{override(\mathtt{C},\mathtt{m})}$$

$$\text{WFC}\ \frac{\begin{array}{c}\overline{\texttt{X N}}; \_ \vdash \overline{\texttt{N}}, \overline{\texttt{T}}, (\texttt{this}_u\ \texttt{C'<}\overline{\texttt{T'}}\texttt{>})\ \texttt{ok} \\ \overline{\texttt{mt}}\ \texttt{ok in}\ \ \texttt{C<}\overline{\texttt{X N}}\texttt{>} \qquad \texttt{rep} \notin \overline{\texttt{N}}\end{array}}{\texttt{class C<}\overline{\texttt{X N}}\texttt{> extends C'<}\overline{\texttt{T'}}\texttt{>}\ \{\ \overline{\texttt{f T}}\texttt{;}\ \overline{\texttt{mt}}\ \}\ \texttt{ok}}$$

$$\text{WFP}\ \frac{\begin{array}{c}\overline{\texttt{Cls}}\ \texttt{ok} \\ \texttt{class}\ \ \texttt{C<>}\ldots\ \in \overline{\texttt{Cls}} \\ \epsilon;\ \texttt{this}\ (\texttt{this}_u\ \texttt{C<>}) \vdash \texttt{e} : \texttt{N}\end{array}}{\overline{\texttt{Cls}}, \texttt{C}, \texttt{e ok}} \qquad \text{SWFE}\ \frac{\begin{array}{c}\Gamma = \overline{\texttt{X N}}, \overline{\texttt{X' N'}}\ ; \\ \texttt{this}\ (\texttt{this}_u\ \texttt{C<}\overline{\texttt{X}}\texttt{>}), \overline{\texttt{x T}} \\ \texttt{class C<}\overline{\texttt{X N}}\texttt{>}\ \ldots \\ \Gamma \vdash \overline{\texttt{N}}, \overline{\texttt{N'}}, \overline{\texttt{T}}\ \texttt{ok}\end{array}}{\Gamma\ \texttt{ok}}$$

Fig. 9. Well-formedness rules

Well-Formed Types. The judgment $\Gamma \vdash \texttt{T ok}$ expresses that type `T` is well-formed in type environment Γ. Type variables are well-formed, if they are contained in the type environment (WFT-1). A non-variable type $\texttt{u C<}\overline{\texttt{T}}\texttt{>}$ is well-formed if its type arguments $\overline{\texttt{T}}$ are well-formed and for each type parameter the actual type argument is a subtype of the upper bound, adapted from the viewpoint $\texttt{u C<}\overline{\texttt{T}}\texttt{>}$ (WFT-2). The viewpoint adaptation is necessary because the type arguments describe ownership relative to the `this` object where $\texttt{u C<}\overline{\texttt{T}}\texttt{>}$ is used, whereas the upper bounds are relative to the object of type $\texttt{u C<}\overline{\texttt{T}}\texttt{>}$. Note that rule WFT-2 permits type variables of a class `C` to be used in upper bounds of `C`. For instance in class `IterImpl` (Fig. 4), type variable `X` is used in its own upper bound, `any MapNode<K, V, X>`.

Well-Formed Methods. The judgment $\texttt{mt ok in C<}\overline{\texttt{X N}}\texttt{>}$ expresses that method `mt` is well-formed in a class `C` with type parameters $\overline{\texttt{X N}}$. According to rule WFM-1, `mt` is well-formed if: (1) the return type, the upper bounds of `mt`'s type variables, and `mt`'s parameter types are well-formed in the type environment that maps `mt`'s and `C`'s type variables to their upper bounds as well as `this` and the explicit method parameters to their types. The type of `this` is the enclosing class, $\texttt{C<}\overline{\texttt{X}}\texttt{>}$, with main modifier $\texttt{this}_u$; (2) the method body, expression `e`, is well-typed with `mt`'s return type; (3) `mt` respects the rules for overriding, see below; (4) if `mt` is pure then the only ownership modifier that occurs in a parameter type or the upper bound of a method type variable is `any`. We will motivate the fourth requirement when we explain the type rule for method calls.

Method `m` respects the rules for overriding if it does not override a method or if all overridden methods have the identical signatures after substituting type variables of the superclasses by the instantiations given in the subclass (WFM-2). For simplicity, we require that overrides do not change the purity of a method, although overriding non-pure methods by pure methods would be safe.

Well-Formed Classes. The judgment `Cls ok` expresses that class declaration `Cls` is well-formed. According to rule WFC, this is the case if: (1) the upper bounds of `Cls`'s type variables, the types of `Cls`'s fields, and the instantiation of the superclass are well-formed in the type environment that maps `Cls`'s type variables to their upper bounds; (2) `Cls`'s methods are well-formed; (3) `Cls`'s upper bounds do not contain the `rep` modifier.

Note that `Cls`'s upper bounds express ownership relative to the current `Cls` instance. If such an upper bound contains a `rep` modifier, clients of `Cls` cannot instantiate `Cls`. The ownership modifiers of an actual type argument are relative to the client's viewpoint. From this viewpoint, none of the modifiers `peer`, `rep`, or `any` expresses that an object is owned by the `Cls` instance. Therefore, we forbid upper bounds with `rep` modifiers by Requirement (3).

Well-Formed Programs. The judgment `P ok` expresses that program `P` is well-formed. According to rule WFP, this holds if all classes in `P` are well-formed, the main class `C` is a non-generic class in `P`, and the main expression `e` is

well-typed in an environment with `this` as an instance of `C`. We omit checks for valid appearances of the ownership modifier $\texttt{this}_u$. As explained earlier, $\texttt{this}_u$ must not occur in the program.

Well-Formed Type Environments. The judgment $\Gamma\ \texttt{ok}$ expresses that type environment Γ is well-formed. According to rule SWFE, this is the case if all upper bounds of type variables and the types of method parameters are well-formed. Moreover, `this` must be mapped to a non-variable type with main modifier $\texttt{this}_u$ and an uninstantiated class.

3.6 Type Rules

We are now ready to present the type rules (Fig. 10). The judgment $\Gamma \vdash \texttt{e} : \texttt{T}$ expresses that expression `e` is well-typed with type `T` in environment Γ. Our type rules implicitly require types to be well-formed, that is, a type rule is applicable only if all types involved in the rule are well-formed in the respective environment.

$$\text{GT-Subs}\ \frac{\Gamma \vdash \texttt{e} : \texttt{T} \qquad \Gamma \vdash \texttt{T} <: \texttt{T}'}{\Gamma \vdash \texttt{e} : \texttt{T}'} \qquad \text{GT-Var}\ \frac{\texttt{x} \in \mathrm{dom}(\Gamma)}{\Gamma \vdash \texttt{x} : \Gamma(\texttt{x})} \qquad \text{GT-Null}\ \frac{\texttt{T} \neq \texttt{this}_u\ \texttt{\_<\_>}}{\Gamma \vdash \texttt{null} : \texttt{T}}$$

$$\text{GT-New}\ \frac{\texttt{N} \neq \texttt{any}_u\ \texttt{\_<\_>}}{\Gamma \vdash \texttt{new N} : \texttt{N}} \qquad \text{GT-Cast}\ \frac{\Gamma \vdash \texttt{e}_0 : \texttt{T}_0}{\Gamma \vdash \texttt{(T) e}_0 : \texttt{T}}$$

$$\text{GT-Read}\ \frac{\begin{array}{c}\Gamma \vdash \texttt{e}_0 : \texttt{N}_0 \\ \texttt{N}_0 = \texttt{\_ C}_0\texttt{<\_>} \\ \texttt{T}_1 = \mathit{fType}(\texttt{C}_0, \texttt{f})\end{array}}{\Gamma \vdash \texttt{e}_0\texttt{.f} : \texttt{N}_0 \triangleright \texttt{T}_1} \qquad \text{GT-Upd}\ \frac{\begin{array}{c}\Gamma \vdash \texttt{e}_0 : \texttt{N}_0 \qquad \texttt{N}_0 = \texttt{u}_0\ \texttt{C}_0\texttt{<\_>} \\ \texttt{T}_1 = \mathit{fType}(\texttt{C}_0, \texttt{f}) \\ \Gamma \vdash \texttt{e}_2 : \texttt{N}_0 \triangleright \texttt{T}_1 \\ \texttt{u}_0 \neq \texttt{any} \qquad \mathit{rp}(\texttt{u}_0, \texttt{T}_1)\end{array}}{\Gamma \vdash \texttt{e}_0\texttt{.f=e}_2 : \texttt{N}_0 \triangleright \texttt{T}_1}$$

$$\text{GT-Invk}\ \frac{\begin{array}{c}\Gamma \vdash \texttt{e}_0 : \texttt{N}_0 \qquad \texttt{N}_0 = \texttt{u}_0\ \texttt{C}_0\texttt{<\_>} \\ \mathit{mType}(\texttt{C}_0, \texttt{m}) = \texttt{<}\overline{\texttt{X}_m\ \texttt{N}_b}\texttt{>}\ \texttt{w}\ \texttt{T}_r\ \texttt{m(}\overline{\texttt{x}\ \texttt{T}_p}\texttt{)} \\ \Gamma \vdash \overline{\texttt{T}} <: (\texttt{N}_0 \triangleright \overline{\texttt{N}_b})[\overline{\texttt{T}/\texttt{X}_m}] \qquad \Gamma \vdash \overline{\texttt{e}_2} : (\texttt{N}_0 \triangleright \overline{\texttt{T}_p})[\overline{\texttt{T}/\texttt{X}_m}] \\ (\texttt{u}_0 = \texttt{any} \Rightarrow \texttt{w} = \texttt{pure}) \qquad \mathit{rp}(\texttt{u}_0, \overline{\texttt{T}_p} \circ \overline{\texttt{N}_b})\end{array}}{\Gamma \vdash \texttt{e}_0\texttt{.m<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}_2}\texttt{)} : (\texttt{N}_0 \triangleright \texttt{T}_r)[\overline{\texttt{T}/\texttt{X}_m}]}$$

Fig. 10. Type rules

An expression of type `T` can also be typed with `T`'s supertypes (GT-Subs). The type of method parameters (including `this`) is determined by a lookup in the type environment (GT-Var). The `null`-reference can have any type other than a $\texttt{this}_u$ type (GT-Null). Objects must be created in a specific context. Therefore only non-variable types with an ownership modifier other than $\texttt{any}_u$ are allowed for object creations (GT-New). The rule for casts (GT-Cast) is straightforward; it could be strengthened to prevent more cast errors statically, but we omit this check since it is not strictly needed.

As explained in detail in Sec. 3.2, the type of a field access is determined by adapting the declared type of the field from the viewpoint described by the type of the receiver (GT-Read). If this type is a type variable, subsumption is used to go to its upper bound because *fType* is defined on class identifiers. Subsumption is also used for inherited fields to ensure that `f` is actually declared in $\mathtt{C}_0$. (Recall that $\mathit{fType}(\mathtt{C}_0, \mathtt{f})$ is undefined otherwise.)

For a field update, the right-hand side expression must be typable as the viewpoint-adapted field type, which is also the type of the whole field update expression (GT-Upd). The rule is analogous to field read, but has two additional requirements. First, the main modifier $\mathtt{u}_0$ of the type of the receiver expression must not be `any`. With the owner-as-modifier discipline, a method must not update fields of objects in arbitrary contexts. Second, the requirement $rp(\mathtt{u}_0, \mathtt{T}_1)$ enforces that `f` is updated through receiver `this` if its declared type $\mathtt{T}_1$ contains a `rep` modifier. For all other receivers, the viewpoint adaptation $\mathtt{N}_0 \triangleright \mathtt{T}_1$ yields an `any` type, but it is obviously unsafe to update `f` with an object with an arbitrary owner. It is convenient to define *rp* for sequences of types. The definition uses the fact that the ownership modifier $\mathtt{this}_u$ is only used for the type of `this`:

$$\begin{aligned} rp :: \mathtt{OM} \times \overline{^{\mathtt{s}}\mathtt{Type}} &\to \mathit{bool} \\ rp(\mathtt{u}, \overline{\mathtt{T}}) &= \mathtt{u} = \mathtt{this}_u \vee (\forall i : \mathtt{rep} \notin \mathtt{T}_i) \end{aligned}$$

The rule for method calls (GT-Invk) is in many ways similar to field reads (for result passing) and updates (for argument passing). The method signature is determined using the receiver type $\mathtt{N}_0$ and subsumption. The type of the invocation expression is determined by viewpoint adaptation of the return type $\mathtt{T}_r$ from the receiver type $\mathtt{N}_0$. Modulo subsumption, the actual method arguments must have the formal parameter types, adapted from $\mathtt{N}_0$ and with actual type arguments $\overline{\mathtt{T}}$ substituted for the method's type variables $\mathtt{X}_m$. For instance, in the call `first.init(key, value, first)` in method `put` (Fig. 2), the adapted third formal parameter type is `rep Node<K,V>` $\triangleright$ `peer Node<K,V>` (note that `Node` substitutes the type variable `X` by `peer Node<K,V>`). This adaptation yields `rep Node<K,V>`, which is also the type of the third actual method argument.

To enforce the owner-as-modifier discipline, only pure methods may be called on receivers with main modifier `any`. For a call on a receiver with main modifier `any`, the viewpoint-adapted formal parameter type contains only the modifier `any`. Consequently, arguments with arbitrary owners can be passed. For this to be type safe, pure methods must not expect arguments with specific owners. This is enforced by rule WFM-1 (Fig. 9). Finally, if the receiver is different from `this`, then neither the formal parameter types nor the upper bounds of the method's type variables must contain `rep`.

4 Runtime Model

In this section, we explain the runtime model of Generic Universe Types. We present the heap model, the runtime type information, well-formedness conditions, and an operational semantics.

4.1 Heap Model

Fig. 11 defines our model of the heap. The prefix $^{\mathtt{r}}$ distinguishes sorts of the runtime model from their static counterparts.

$$
\begin{array}{rcl}
\mathtt{h} \in \mathtt{Heap} & = & \mathtt{Addr} \rightarrow \mathtt{Obj} \\
\iota \in \mathtt{Addr} & = & \text{Address} \mid \mathtt{null}_a \\
\mathtt{o} \in \mathtt{Obj} & = & {}^{\mathtt{r}}\mathtt{T},\ \mathtt{Fs} \\
{}^{\mathtt{r}}\mathtt{T} \in {}^{\mathtt{r}}\mathtt{Type} & = & \iota_o\ \mathtt{C}\texttt{<}\overline{{}^{\mathtt{r}}\mathtt{T}}\texttt{>} \\
\mathtt{Fs} \in \mathtt{Fields} & = & \mathtt{FieldId} \rightarrow \mathtt{Addr} \\
\iota_o \in \mathtt{OwnerAddr} & = & \iota \mid \mathtt{any}_a \\
{}^{\mathtt{r}}\Gamma \in {}^{\mathtt{r}}\mathtt{Env} & = & \overline{\mathtt{X}\ {}^{\mathtt{r}}\mathtt{T}};\ \overline{\mathtt{x}\ \iota}
\end{array}
$$

Fig. 11. Definitions for the heap model

A heap ($\mathtt{h} \in \mathtt{Heap}$) maps addresses to objects. An address ($\iota \in \mathtt{Addr}$) can be the special null-reference $\mathtt{null}_a$. An object ($\mathtt{o} \in \mathtt{Obj}$) consist of its runtime type and a mapping from field identifiers to the addresses stored in the fields.

The runtime type ($^{\mathtt{r}}\mathtt{T} \in {}^{\mathtt{r}}\mathtt{Type}$) of an object o consists of the address of o's owner object, of o's class, and of runtime types for the type arguments of this class. We store the runtime type arguments including the associated ownership information explicitly in the heap because this information is needed in the runtime checks for casts. In that respect, our runtime model is similar to that of the .NET CLR [16]. The owner address of objects in the root context is $\mathtt{null}_a$. The special owner address $\mathtt{any}_a$ is used when the corresponding static type has the $\mathtt{any}_u$ modifier. Consider for instance an execution of method main (Fig. 5), where the address of this is 1. The runtime type of the object stored in map is $1\ \mathtt{Map<1\ ID}, \mathtt{any}_a\ \mathtt{Data>}$. For simplicity we drop the subscript o from ι_o whenever it is clear from context whether we refer to an Addr or an OwnerAddr.

The first component of a runtime environment ($^{\mathtt{r}}\Gamma \in {}^{\mathtt{r}}\mathtt{Env}$) maps method type variables to their runtime types. The second component is the stack, which maps method parameters to the addresses they store.

Subtyping on Runtime Types. Judgment $\mathtt{h}, \iota \vdash {}^{\mathtt{r}}\mathtt{T} <: {}^{\mathtt{r}}\mathtt{T}'$ expresses that the runtime type $^{\mathtt{r}}\mathtt{T}$ is a subtype of $^{\mathtt{r}}\mathtt{T}'$ from the viewpoint of address ι. The viewpoint, ι, is required in order to give meaning to the ownership modifier rep. Subtyping for runtime types is defined in Fig. 12. Subtyping is transitive (RT-3), and allows owner-invariant (RT-1) and covariant subtyping (RT-2).

Rule RTL introduces owner-invariant subtyping $<:_1$ and defines how subtyping follows subclassing if (1) the runtime types have the same owner address ι', (2) in the type arguments, the ownership modifiers $\mathtt{this}_u$ and peer are substituted by the owner address ι' of the runtime types (we use the same owner address for both modifiers since they both express ownership by the owner of this), (3) rep is substituted by the viewpoint address ι, (4) $\mathtt{any}_u$ is substituted by $\mathtt{any}_a$, (5) the type variables $\overline{\mathtt{X}}$ of the subclass C are substituted consistently by

$\overline{{}^{\mathtt{r}}\mathtt{T}}$, and (6) either the owner of ι is ι' or `rep` does not appear in the instantiation of the superclass. This ensures that the substitution of ι for `rep`-modifiers is meaningful. Note that in a well-formed program, $\mathtt{this}_u$ never occurs in a type argument; nevertheless we include the substitution for consistency. Rule RTL gives the most concrete runtime type deducible from static subclassing.

$$\text{RT-1}\ \frac{\mathtt{h}, \iota \vdash {}^{\mathtt{r}}\mathtt{T} <:_1 {}^{\mathtt{r}}\mathtt{T}'}{\mathtt{h}, \iota \vdash {}^{\mathtt{r}}\mathtt{T} <: {}^{\mathtt{r}}\mathtt{T}'} \qquad \text{RT-2}\ \frac{{}^{\mathtt{r}}\mathtt{T} <:_{\mathtt{a}} {}^{\mathtt{r}}\mathtt{T}'}{\mathtt{h}, \iota \vdash {}^{\mathtt{r}}\mathtt{T} <: {}^{\mathtt{r}}\mathtt{T}'} \qquad \text{RT-3}\ \frac{\mathtt{h}, \iota \vdash {}^{\mathtt{r}}\mathtt{T} <: {}^{\mathtt{r}}\mathtt{T}'' \quad \mathtt{h}, \iota \vdash {}^{\mathtt{r}}\mathtt{T}'' <: {}^{\mathtt{r}}\mathtt{T}'}{\mathtt{h}, \iota \vdash {}^{\mathtt{r}}\mathtt{T} <: {}^{\mathtt{r}}\mathtt{T}'}$$

$$\text{RTL}\ \frac{\mathtt{C{<}\overline{X}{>}} \sqsubseteq \mathtt{C}'\mathtt{{<}\overline{{}^{s}T}{>}} \quad \mathrm{dom}(\mathtt{C}) = \overline{\mathtt{X}} \quad \mathit{owner}(\mathtt{h}, \iota) = \iota' \ \vee\ \mathtt{rep} \notin \overline{{}^{\mathtt{s}}\mathtt{T}}}{\mathtt{h}, \iota \vdash \iota'\ \mathtt{C{<}\overline{{}^{r}T}{>}} <:_1 \iota'\ \mathtt{C}'\mathtt{{<}\overline{{}^{s}T}[\iota'/\mathtt{this}_u, \iota'/\mathtt{peer}, \iota/\mathtt{rep}, \mathtt{any}_a/\mathtt{any}_u, \overline{{}^{r}T/X}]{>}}}$$

$$\text{RTA-1}\ \frac{}{{}^{\mathtt{r}}\mathtt{T} <:_{\mathtt{a}} {}^{\mathtt{r}}\mathtt{T}} \qquad \text{RTA-2}\ \frac{\overline{{}^{\mathtt{r}}\mathtt{T} <:_{\mathtt{a}} {}^{\mathtt{r}}\mathtt{T}'}}{\iota'\ \mathtt{C{<}\overline{{}^{r}T}{>}} <:_{\mathtt{a}} \mathtt{any}_a\ \mathtt{C{<}\overline{{}^{r}T'}{>}}} \qquad \text{RTH-1}\ \frac{h(\iota) = {}^{\mathtt{r}}\mathtt{T}, \_ \quad \mathtt{h}, \iota \vdash {}^{\mathtt{r}}\mathtt{T} <: {}^{\mathtt{r}}\mathtt{T}'}{\mathtt{h} \vdash \iota : {}^{\mathtt{r}}\mathtt{T}'}$$

$$\text{RTH-2}\ \frac{}{\mathtt{h} \vdash \mathtt{null}_a : {}^{\mathtt{r}}\mathtt{T}'} \qquad \text{RTS}\ \frac{\mathtt{h} \vdash \iota : \mathit{dyn}({}^{\mathtt{s}}\mathtt{T}, \mathtt{h}, {}^{\mathtt{r}}\Gamma) \quad {}^{\mathtt{s}}\mathtt{T} = \mathtt{this}_u\ \mathtt{\_{<}\_{>}} \Rightarrow \iota = {}^{\mathtt{r}}\Gamma(\mathtt{this})}{\mathtt{h}, {}^{\mathtt{r}}\Gamma \vdash \iota : {}^{\mathtt{s}}\mathtt{T}}$$

Fig. 12. Rules for subtyping on runtime types

As for subtyping for static types, we have limited covariance for runtime types. Covariant subtyping is expressed by the relation $<:_{\mathtt{a}}$. The rules for limited covariance, RTA-1 and RTA-2, are analogous to the rules TA-1 and TA-2 for static types (Fig. 8). Reflexivity of $<:$ follows from RTA-1 and RT-2.

The judgment $\mathtt{h} \vdash \iota : {}^{\mathtt{r}}\mathtt{T}'$ expresses that in heap `h`, the address ι has type ${}^{\mathtt{r}}\mathtt{T}'$. The type of ι is determined by the type of the object at ι and the subtype relation (RTH-1). The `null` reference can have any type (RTH-2).

Finally, the judgment $\mathtt{h}, {}^{\mathtt{r}}\Gamma \vdash \iota : {}^{\mathtt{s}}\mathtt{T}$ expresses that in heap `h` and runtime environment ${}^{\mathtt{r}}\Gamma$, the address ι has a runtime type that corresponds to the static type ${}^{\mathtt{s}}\mathtt{T}$ (see below for the definition of *dyn*) and that the main modifier $\mathtt{this}_u$ is used solely for the type of `this` (RTS).

From Static Types to Runtime Types. Static types and runtime types are related by the following *dynamization function*, which is defined by rule DYN:

$$\mathit{dyn} :: {}^{\mathtt{s}}\mathtt{Type} \times \mathtt{Heap} \times {}^{\mathtt{r}}\mathtt{Env} \to {}^{\mathtt{r}}\mathtt{Type}$$

$$\text{DYN}\ \frac{{}^{\mathtt{r}}\Gamma = \overline{\mathtt{X}'\ {}^{\mathtt{r}}\mathtt{T}'};\ \mathtt{this}\ \iota, \_ \quad \mathrm{dom}(\mathtt{C}) = \overline{\mathtt{X}} \qquad \mathtt{h}, \iota \vdash \mathtt{h}(\iota)\downarrow_1 <:_1 \iota'\ \mathtt{C{<}\overline{{}^{r}T}{>}} \quad \mathrm{free}({}^{\mathtt{s}}\mathtt{T}) \subseteq \overline{\mathtt{X}} \circ \overline{\mathtt{X}'}}{\mathit{dyn}({}^{\mathtt{s}}\mathtt{T}, \mathtt{h}, {}^{\mathtt{r}}\Gamma) = {}^{\mathtt{s}}\mathtt{T}[\iota'/\mathtt{this}, \iota'/\mathtt{peer}, \iota/\mathtt{rep}, \mathtt{any}_a/\mathtt{any}_u, \overline{{}^{r}T/X}, \overline{{}^{r}T'/X'}]}$$

This function maps a static type ${}^{\mathtt{s}}\mathtt{T}$ to the corresponding runtime type. The viewpoint is described by a heap `h` and a runtime environment ${}^{\mathtt{r}}\Gamma$. In ${}^{\mathtt{s}}\mathtt{T}$, *dyn*

substitutes `rep` by the address of the `this` object (ι), `peer` and $\texttt{this}_u$ by the owner of ι (ι'), and $\texttt{any}_u$ by $\texttt{any}_a$. It also substitutes all type variables in ${}^{\texttt{s}}\texttt{T}$ by the instantiations given in ι' $\texttt{C<}\overline{{}^{\texttt{r}}\texttt{T}}\texttt{>}$, a supertype of ι's runtime type, or in the runtime environment. The substitutions performed by *dyn* are analogous to the ones in rule RTL (Fig. 12), which also involves mapping static types to runtime types. We do not use *dyn* in RTL to avoid that the definitions of $<:$ and *dyn* are mutually recursive. We use projection $\downarrow_i$ to select the i-th component of a tuple, for instance, the runtime type and field mapping of an object.

Note that the outcome of *dyn* depends on finding ι' $\texttt{C<}\overline{{}^{\texttt{r}}\texttt{T}}\texttt{>}$, an appropriate supertype of the runtime type of the `this` object ι, which contains substitutions for all type variables not mapped by the environment (free(${}^{\texttt{s}}\texttt{T}$) yields the free type variables in ${}^{\texttt{s}}\texttt{T}$). Thus, one may wonder whether there is more than one such appropriate superclass. However, because type variables are globally unique, if the free variables of ${}^{\texttt{s}}\texttt{T}$ are in the domain of a class then they are not in the domain of any other class. To obtain the most precise ownership information we use the owner-invariant runtime subtype relation $<:_{\texttt{1}}$ defined in rule RTL.

To illustrate dynamization, consider an execution of `put` (Fig. 2), in an environment ${}^{\texttt{r}}\Gamma$ whose `this` object has address 3 and a heap `h` where address 3 has runtime type 1 $\texttt{Map<1 ID,}\,\texttt{any}_a\,\texttt{Data>}$ (see Fig. 1). We determine the runtime type of the object created by `new rep Node<K,V>`. The dynamization of the type of the new object w.r.t. `h` and ${}^{\texttt{r}}\Gamma$ is $dyn(\texttt{rep Node<K,V>}, \texttt{h}, {}^{\texttt{r}}\Gamma)$, which yields 3 $\texttt{Node<1 ID,}\,\texttt{any}_a\,\texttt{Data>}$. This runtime type correctly reflects that the new object is owned by `this` (owner address 3) and has the same type arguments as the runtime type of `this`.

It is convenient to define the following overloaded version of *dyn*:

$$dyn({}^{\texttt{s}}\texttt{T}, \texttt{h}, \iota) = dyn({}^{\texttt{s}}\texttt{T}, \texttt{h}, (\epsilon; \texttt{this}\ \iota))$$

4.2 Lookup Functions

In this subsection, we define the functions to look up the runtime type of a field or the body of a method.

Field Lookup. The runtime type of a field `f` is essentially the dynamization of its static type. The function ${}^{\texttt{r}}fType(\texttt{h}, \iota, \texttt{f})$ yields the runtime type of `f` in an object at address ι in heap `h`. In its definition (RFT, in Fig. 13), `C` is the runtime class of ι, and C' is the superclass of `C` which contains the definition of `f`.

Method Lookup. The function $mBody(\texttt{C}, \texttt{m})$ yields a tuple consisting of `m`'s body expression as well as the identifiers of its formal parameters and type variables. This is trivial if `m` is declared in `C` (RMT-1, Fig. 13). Otherwise, `m` is looked up in `C`'s superclass $\texttt{C}'$ (RMT-2).

4.3 Well-Formedness

In this subsection, we define well-formedness of runtime types, heaps, and runtime environments. The rules are presented in Fig. 13.

$$\text{RFT}\ \frac{\mathtt{h}(\iota)\downarrow_1 = \_\ \mathtt{C}\texttt{<}\_\texttt{>} \qquad \mathtt{C}\texttt{<}\_\texttt{>} \sqsubseteq \mathtt{C}'\texttt{<}\_\texttt{>}}{{}^{\mathtt{r}}fType(\mathtt{h},\iota,\mathtt{f}) = dyn({}^{\mathtt{s}}fType(\mathtt{C}',\mathtt{f}),\mathtt{h},\iota)}$$

$$\text{RMT-1}\ \frac{\texttt{class C<\_> extends \_<\_> \{ \_ ; ... <}\overline{\texttt{X \_}}\texttt{> \_ \_ m(}\overline{\texttt{x \_}}\texttt{) \{ return e \} ... \}}}{mBody(\mathtt{C},\mathtt{m}) = (\mathtt{e},\overline{\mathtt{x}},\overline{\mathtt{X}})}$$

$$\text{RMT-2}\ \frac{\texttt{class C<\_> extends C'<\_> \{ no method m \}}}{mBody(\mathtt{C},\mathtt{m}) = mBody(\mathtt{C}',\mathtt{m})}$$

$$\text{WFRT}\ \frac{\begin{array}{c}\iota' \in \mathrm{dom}(\mathtt{h}) \cup \{\mathtt{null}_a, \mathtt{any}_a\} \qquad \mathtt{h},\iota \vdash \overline{{}^{\mathtt{r}}\mathtt{T}}\ \mathtt{ok} \\ \texttt{class C<\_}\ \overline{{}^{\mathtt{s}}\mathtt{N}}\texttt{>}\ \ldots \qquad \mathtt{h},\iota \vdash \overline{{}^{\mathtt{r}}\mathtt{T}} <: dyn(\overline{{}^{\mathtt{s}}\mathtt{N}},\mathtt{h},\iota)\end{array}}{\mathtt{h},\iota \vdash \iota'\ \texttt{C<}\overline{{}^{\mathtt{r}}\mathtt{T}}\texttt{>}\ \mathtt{ok}}$$

$$\text{WFH}\ \frac{\begin{array}{c}\mathtt{null}_a \notin \mathrm{dom}(\mathtt{h}) \qquad \forall\iota:\ \mathtt{h},\iota \vdash \mathtt{h}(\iota)\downarrow_1\ \mathtt{ok} \ \wedge\ \mathtt{null}_a \in owners(\mathtt{h},\iota) \\ \forall\iota,\mathtt{f}:\ \mathtt{h}(\iota)\downarrow_2 = \mathtt{Fs} \ \wedge\ {}^{\mathtt{r}}fType(\mathtt{h},\iota,\mathtt{f}) = {}^{\mathtt{r}}\mathtt{T} \implies \mathtt{h} \vdash \mathtt{Fs}(\mathtt{f}) : {}^{\mathtt{r}}\mathtt{T}\end{array}}{\mathtt{h}\ \mathtt{ok}}$$

$$\text{WFRE}\ \frac{\begin{array}{c}{}^{\mathtt{r}}\Gamma = \overline{\mathtt{X}\ {}^{\mathtt{r}}\mathtt{T}}\ ;\ \mathtt{this}\ \iota, \overline{\mathtt{x}\ \iota'} \qquad {}^{\mathtt{s}}\Gamma = \overline{\mathtt{X}\ {}^{\mathtt{s}}\mathtt{N}},\ \overline{\mathtt{X}'\ \_}\ ;\ \mathtt{this}\ (\mathtt{this}_u\ \texttt{C<}\overline{\mathtt{X}}\texttt{>}), \overline{\mathtt{x}\ {}^{\mathtt{s}}\mathtt{T}'} \\ \mathtt{h}\ \mathtt{ok} \quad {}^{\mathtt{s}}\Gamma\ \mathtt{ok} \quad \iota \neq \mathtt{null}_a \quad \mathtt{h},{}^{\mathtt{r}}\Gamma \vdash \overline{{}^{\mathtt{r}}\mathtt{T}}\ \mathtt{ok} \quad \mathtt{h},{}^{\mathtt{r}}\Gamma \vdash \overline{{}^{\mathtt{r}}\mathtt{T}} <: \overline{dyn({}^{\mathtt{s}}\mathtt{N},\mathtt{h},{}^{\mathtt{r}}\Gamma)} \\ \mathtt{h},{}^{\mathtt{r}}\Gamma \vdash \iota :\ \mathtt{this}_u\ \texttt{C<}\overline{\mathtt{X}'}\texttt{>} \qquad \mathtt{h},{}^{\mathtt{r}}\Gamma \vdash \overline{\iota'} :\ \overline{{}^{\mathtt{s}}\mathtt{T}'}\end{array}}{\mathtt{h} \vdash {}^{\mathtt{r}}\Gamma : {}^{\mathtt{s}}\Gamma}$$

Fig. 13. Rules for field and method lookup, and well-formedness

Well-Formed Runtime Types. The judgment $\mathtt{h},\iota \vdash \iota'\ \texttt{C<}\overline{{}^{\mathtt{r}}\mathtt{T}}\texttt{>}\ \mathtt{ok}$ expresses that runtime type $\iota'\ \texttt{C<}\overline{{}^{\mathtt{r}}\mathtt{T}}\texttt{>}$ is well-formed for viewpoint address ι in heap $\mathtt{h}$. According to rule WFRT, the owner address ι' must be the address of an object in the heap $\mathtt{h}$ or one of the special owners $\mathtt{null}_a$ and $\mathtt{any}_a$. All type arguments must also be well-formed types. A runtime type must have a type argument for each type variable of its class. Each runtime type argument must be a subtype of the dynamization of the type variable's upper bound. We use $\mathtt{h},{}^{\mathtt{r}}\Gamma \vdash {}^{\mathtt{r}}\mathtt{T}\ \mathtt{ok}$ as shorthand for $\mathtt{h},{}^{\mathtt{r}}\Gamma(\mathtt{this}) \vdash {}^{\mathtt{r}}\mathtt{T}\ \mathtt{ok}$.

Well-Formed Heaps. A heap $\mathtt{h}$ is well-formed, denoted by $\mathtt{h}\ \mathtt{ok}$, if and only if the $\mathtt{null}_a$ address is not mapped to an object, the runtime types of all objects are well-formed, the root owner $\mathtt{null}_a$ is in the set of owners of all objects, and all addresses stored in fields are well-typed (WFH). By mandating that all objects are (transitively) owned by $\mathtt{null}_a$ and because each runtime type has one unique owner address, we ensure that ownership is a tree structure.

Well-Formed Runtime Environments. The judgment $\mathtt{h} \vdash {}^{\mathtt{r}}\Gamma : {}^{\mathtt{s}}\Gamma$ expresses that runtime environment ${}^{\mathtt{r}}\Gamma$ is well-formed w.r.t. a well-formed heap $\mathtt{h}$ and a well-formed static type environment ${}^{\mathtt{s}}\Gamma$. According to rule WFRE, this is the case if and only if: (1) ${}^{\mathtt{r}}\Gamma$ maps all method type variables $\overline{\mathtt{X}}$ that are contained in ${}^{\mathtt{s}}\Gamma$ to well-formed runtime types $\overline{{}^{\mathtt{r}}\mathtt{T}}$, which are subtypes of the dynamizations of the corresponding upper bounds $\overline{{}^{\mathtt{s}}\mathtt{N}}$; (2) ${}^{\mathtt{r}}\Gamma$ maps $\mathtt{this}$ to an address ι. The object at address ι is well-typed with the static type of $\mathtt{this}$, $\mathtt{this}_u\ \texttt{C<}\overline{\mathtt{X}'}\texttt{>}$. (3) ${}^{\mathtt{r}}\Gamma$ maps

the formal parameters $\overline{\mathtt{x}}$ that are contained in ${}^{\mathtt{s}}\Gamma$ to addresses $\overline{\iota'}$. The objects at addresses $\overline{\iota'}$ are well-typed with the static types of $\overline{\mathtt{x}}$, $\overline{{}^{\mathtt{s}}\mathtt{T}'}$.

4.4 Operational Semantics

We describe program execution by a big-step operational semantics. The transition $\mathtt{h}, {}^{\mathtt{r}}\Gamma, \mathtt{e} \rightsquigarrow \mathtt{h}', \iota$ expresses that the evaluation of an expression $\mathtt{e}$ in heap $\mathtt{h}$ and runtime environment ${}^{\mathtt{r}}\Gamma$ results in address ι and successor heap $\mathtt{h}'$. A program with main class $\mathtt{C}$ is executed by evaluating the main expression in a heap $\mathtt{h}_0$ that contains exactly one $\mathtt{C}$ instance in the root context where all fields $\overline{\mathtt{f}}$ are initialized to $\mathtt{null}_a$ ($\mathtt{h}_0 = \{\iota \mapsto (\mathtt{null}_a\ \mathtt{C}\texttt{<>},\ \overline{\mathtt{f}\ \mathtt{null}_a})\}$) and a runtime environment ${}^{\mathtt{r}}\Gamma_0$ that maps `this` to this $\mathtt{C}$ instance (${}^{\mathtt{r}}\Gamma_0 = \epsilon\texttt{;this}\ \iota$). The rules for evaluating expressions are presented in Fig. 14 and explained in the following.

$$\text{OS-Var}\ \frac{}{\mathtt{h}, {}^{\mathtt{r}}\Gamma, \mathtt{x} \rightsquigarrow \mathtt{h}, {}^{\mathtt{r}}\Gamma(\mathtt{x})} \qquad \text{OS-Null}\ \frac{}{\mathtt{h}, {}^{\mathtt{r}}\Gamma, \mathtt{null} \rightsquigarrow \mathtt{h}, \mathtt{null}_a}$$

$$\text{OS-Cast}\ \frac{\mathtt{h}, {}^{\mathtt{r}}\Gamma, \mathtt{e}_0 \rightsquigarrow \mathtt{h}', \iota \quad \mathtt{h}', {}^{\mathtt{r}}\Gamma \vdash \iota : {}^{\mathtt{s}}\mathtt{T}}{\mathtt{h}, {}^{\mathtt{r}}\Gamma, ({}^{\mathtt{s}}\mathtt{T})\ \mathtt{e}_0 \rightsquigarrow \mathtt{h}', \iota} \qquad \text{OS-New}\ \frac{\iota \notin \mathrm{dom}(h) \quad \iota \neq \mathtt{null}_a \quad {}^{\mathtt{r}}\mathtt{T} = \mathit{dyn}({}^{\mathtt{s}}\mathtt{N}, \mathtt{h}, {}^{\mathtt{r}}\Gamma) = \_\ \mathtt{C}\texttt{<\_>} \quad \mathtt{Fs}(\mathit{fields}(\mathtt{C})) = \mathtt{null}_a \quad \mathtt{h}' = \mathtt{h}[\iota \mapsto ({}^{\mathtt{r}}\mathtt{T}, \mathtt{Fs})]}{\mathtt{h}, {}^{\mathtt{r}}\Gamma, \mathtt{new}\ {}^{\mathtt{s}}\mathtt{N} \rightsquigarrow \mathtt{h}', \iota}$$

$$\text{OS-Read}\ \frac{\mathtt{h}, {}^{\mathtt{r}}\Gamma, \mathtt{e}_0 \rightsquigarrow \mathtt{h}', \iota_0 \quad \iota_0 \neq \mathtt{null}_a \quad \iota = \mathtt{h}'(\iota_0)\downarrow_2(\mathtt{f})}{\mathtt{h}, {}^{\mathtt{r}}\Gamma, \mathtt{e}_0.\mathtt{f} \rightsquigarrow \mathtt{h}', \iota} \qquad \text{OS-Upd}\ \frac{\mathtt{h}, {}^{\mathtt{r}}\Gamma, \mathtt{e}_0 \rightsquigarrow \mathtt{h}_0, \iota_0 \quad \iota_0 \neq \mathtt{null}_a \quad \mathtt{h}_0, {}^{\mathtt{r}}\Gamma, \mathtt{e}_2 \rightsquigarrow \mathtt{h}_2, \iota \quad \mathtt{h}' = \mathtt{h}_2[\iota_0.\mathtt{f} := \iota]}{\mathtt{h}, {}^{\mathtt{r}}\Gamma, \mathtt{e}_0.\mathtt{f}{=}\mathtt{e}_2 \rightsquigarrow \mathtt{h}', \iota}$$

$$\text{OS-Invk}\ \frac{\mathtt{h}, {}^{\mathtt{r}}\Gamma, \mathtt{e}_0 \rightsquigarrow \mathtt{h}_0, \iota_0 \quad \iota_0 \neq \mathtt{null}_a \quad \mathtt{h}_0, {}^{\mathtt{r}}\Gamma, \overline{\mathtt{e}_2} \rightsquigarrow \mathtt{h}_2, \overline{\iota_2} \quad \mathtt{h}_0(\iota_0)\downarrow_1 = \_\ \mathtt{C}_0\texttt{<\_>} \quad \mathit{mBody}(\mathtt{C}_0, \mathtt{m}) = (\mathtt{e}_1, \overline{\mathtt{x}}, \overline{\mathtt{X}}) \quad \overline{{}^{\mathtt{r}}\mathtt{T}} = \mathit{dyn}(\overline{{}^{\mathtt{s}}\mathtt{T}}, \mathtt{h}, {}^{\mathtt{r}}\Gamma) \quad {}^{\mathtt{r}}\Gamma' = \overline{\mathtt{X}\ {}^{\mathtt{r}}\mathtt{T}}\ \texttt{;}\ \mathtt{this}\ \iota_0, \overline{\mathtt{x}\ \iota_2} \quad \mathtt{h}_2, {}^{\mathtt{r}}\Gamma', \mathtt{e}_1 \rightsquigarrow \mathtt{h}', \iota}{\mathtt{h}, {}^{\mathtt{r}}\Gamma, \mathtt{e}_0.\mathtt{m}\texttt{<}\overline{{}^{\mathtt{s}}\mathtt{T}}\texttt{>}(\overline{\mathtt{e}_2}) \rightsquigarrow \mathtt{h}', \iota}$$

Fig. 14. Operational semantics

Parameters, including `this`, are evaluated by looking up the stored address in the stack, which is part of the runtime environment ${}^{\mathtt{r}}\Gamma$ (OS-Var). The `null` expression always evaluates to the $\mathtt{null}_a$ address (OS-Null). For cast expressions, we evaluate the expression $\mathtt{e}_0$ and check that the resulting address is well-typed with the static type given in the cast expression w.r.t. the current environment (OS-Cast). Object creation picks a fresh address, allocates an object of the appropriate type, and initializes its fields to $\mathtt{null}_a$ (OS-New). *fields*(C) yields all fields declared in or inherited by C.

For field reads (OS-Read) we evaluate the receiver expression and then look up the field in the heap, provided that the receiver is non-null. For the update of

a field f, we evaluate the receiver expression to address ι_0 and the right-hand side expression to address ι, and update the heap $\mathtt{h}_2$, which is denoted by $\mathtt{h}_2[\iota_0.\mathtt{f} := \iota]$ (OS-Upd). Note that the limited covariance of Generic Universe Types does not require a runtime ownership check for field updates.

For method calls (OS-Invk) we evaluate the receiver expression and actual method arguments in the usual order. The class of the receiver object is used to look up the method body. Its expression is then evaluated in the runtime environment that maps m's type variables to actual type arguments as well as m's formal method parameters (including this) to the actual method arguments. The resulting heap and address are the result of the call. Note that method invocations do not need any runtime type checks or purity checks.

5 Properties

In this section, we present the theorems and proof sketches for type safety and the owner-as-modifier property as well as two important auxiliary lemmas.

Lemmas. The following lemma expresses that viewpoint adaptation from a viewpoint to this is correct. Consider the this object of a runtime environment $^{\mathtt{r}}\Gamma$ and two objects $\mathtt{o}_1$ and $\mathtt{o}_2$. If from the viewpoint this, $\mathtt{o}_1$ has the static type $^{\mathtt{s}}\mathtt{N}$, and from viewpoint $\mathtt{o}_1$, $\mathtt{o}_2$ has the static type $^{\mathtt{s}}\mathtt{T}$, then from the viewpoint this, $\mathtt{o}_2$ has the static type $^{\mathtt{s}}\mathtt{T}$ adapted from $^{\mathtt{s}}\mathtt{N}$, $^{\mathtt{s}}\mathtt{N} \triangleright {}^{\mathtt{s}}\mathtt{T}$. The following lemma expresses this property using the addresses ι_1 and ι_2 of the objects $\mathtt{o}_1$ and $\mathtt{o}_2$, respectively.

Lemma 1 (Adaptation from a Viewpoint)

$$\left.\begin{array}{l} \mathtt{h}, {}^{\mathtt{r}}\Gamma \vdash \iota_1 : {}^{\mathtt{s}}\mathtt{N}, \quad \iota_1 \neq \mathtt{null}_a \\ \mathtt{h}, {}^{\mathtt{r}}\Gamma' \vdash \iota_2 : {}^{\mathtt{s}}\mathtt{T} \\ \mathit{free}({}^{\mathtt{s}}\mathtt{T}) \subseteq \mathit{dom}({}^{\mathtt{s}}\mathtt{N}) \circ \overline{\mathtt{X}} \\ {}^{\mathtt{r}}\Gamma' = \overline{\mathtt{X}}\ \mathit{dyn}(\overline{{}^{\mathtt{s}}\mathtt{T}}, \mathtt{h}, {}^{\mathtt{r}}\Gamma);\ \mathtt{this}\ \iota_1, \_ \end{array}\right\} \implies \mathtt{h}, {}^{\mathtt{r}}\Gamma \vdash \iota_2 : ({}^{\mathtt{s}}\mathtt{N} \triangleright {}^{\mathtt{s}}\mathtt{T})[\overline{{}^{\mathtt{s}}\mathtt{T}/\mathtt{X}}]$$

This lemma justifies the type rule GT-Read. The proof runs by induction on the shape of static type $^{\mathtt{s}}\mathtt{T}$. The base case deals with type variables and non-generic types. The induction step considers generic types, assuming that the lemma holds for the actual type arguments. Each of the cases is done by a case distinction on the main modifiers of $^{\mathtt{s}}\mathtt{N}$ and $^{\mathtt{s}}\mathtt{T}$.

The following lemma is the converse of Lemma 1. It expresses that viewpoint adaptation from this to an object $\mathtt{o}_1$ is correct. If from the viewpoint this, $\mathtt{o}_1$ has the static type $^{\mathtt{s}}\mathtt{N}$ and $\mathtt{o}_2$ has the static type $^{\mathtt{s}}\mathtt{N} \triangleright {}^{\mathtt{s}}\mathtt{T}$, then from viewpoint $\mathtt{o}_1$, $\mathtt{o}_2$ has the static type $^{\mathtt{s}}\mathtt{T}$. The lemma requires that the adaptation of $^{\mathtt{s}}\mathtt{T}$ does not change ownership modifiers in $^{\mathtt{s}}\mathtt{T}$ from non-any to any, because the lost ownership information cannot be recovered. Such a change occurs if $^{\mathtt{s}}\mathtt{N}$'s main modifier is any or if $^{\mathtt{s}}\mathtt{T}$ contains rep and is not accessed through this (see definition of *rp*, Sec. 3.6).

Lemma 2 (Adaptation to a Viewpoint)

$$\left.\begin{array}{l} \mathtt{h}, {}^{\mathtt{r}}\Gamma \vdash \iota_1 : {}^{\mathtt{s}}\mathtt{N}, \quad \iota_1 \neq \mathtt{null}_a \\ \mathtt{h}, {}^{\mathtt{r}}\Gamma \vdash \iota_2 : ({}^{\mathtt{s}}\mathtt{N} \triangleright {}^{\mathtt{s}}\mathtt{T})[\overline{{}^{\mathtt{s}}\mathtt{T}/\mathtt{X}}] \\ {}^{\mathtt{s}}\mathtt{N} = \mathtt{u}\ \mathtt{\_<\_>}, \quad \mathtt{u} \neq \mathtt{any}, \quad rp(\mathtt{u}, {}^{\mathtt{s}}\mathtt{T}) \\ free({}^{\mathtt{s}}\mathtt{T}) \subseteq dom({}^{\mathtt{s}}\mathtt{N}) \circ \overline{\mathtt{X}}, \quad {}^{\mathtt{s}}\mathtt{T} \neq \mathtt{this}_u\ \mathtt{\_<\_>} \\ {}^{\mathtt{r}}\Gamma' = \overline{\mathtt{X}}\ dyn(\overline{{}^{\mathtt{s}}\mathtt{T}}, \mathtt{h}, {}^{\mathtt{r}}\Gamma);\ \mathtt{this}\ \iota_1, \_ \end{array}\right\} \implies \mathtt{h}, {}^{\mathtt{r}}\Gamma' \vdash \iota_2 : {}^{\mathtt{s}}\mathtt{T}$$

This lemma justifies the type rule GT-Upd and the requirements for the types of the parameters in GT-Invk. The proof is analogous to the proof for Lemma 1.

Type Safety for Generic Universe Types is expressed by the following theorem. If a well-typed expression `e` is evaluated in a well-formed environment (including a well-formed heap), then the resulting environment is well-formed and the result of `e`'s evaluation has the type that is the dynamization of `e`'s static type.

Theorem 1 (Type Safety)

$$\left.\begin{array}{l} \mathtt{h} \vdash {}^{\mathtt{r}}\Gamma : {}^{\mathtt{s}}\Gamma \\ {}^{\mathtt{s}}\Gamma \vdash \mathtt{e} : {}^{\mathtt{s}}\mathtt{T} \\ \mathtt{h}, {}^{\mathtt{r}}\Gamma, \mathtt{e} \rightsquigarrow \mathtt{h}', \iota \end{array}\right\} \Longrightarrow \left\{\begin{array}{l} \mathtt{h}' \vdash {}^{\mathtt{r}}\Gamma : {}^{\mathtt{s}}\Gamma \\ \mathtt{h}', {}^{\mathtt{r}}\Gamma \vdash \iota : {}^{\mathtt{s}}\mathtt{T} \end{array}\right.$$

The proof of Theorem 1 runs by rule induction on the operational semantics. Lemma 1 is used to prove field read and method results, whereas Lemma 2 is used to prove field updates and method parameter passing.

We omit a proof of progress since this property is not affected by adding ownership to a Java-like language. The basic proof can be adapted from FGJ [14] and extensions for field updates and casts. The new runtime ownership check in casts can be treated analogously to standard Java casts.

Owner-as-Modifier discipline enforcement is expressed by the following theorem. The evaluation of a well-typed expression `e` in a well-formed environment modifies only those objects that are (transitively) owned by the owner of `this`.

Theorem 2 (Owner-as-Modifier)

$$\left.\begin{array}{l} \mathtt{h} \vdash {}^{\mathtt{r}}\Gamma : {}^{\mathtt{s}}\Gamma \\ {}^{\mathtt{s}}\Gamma \vdash \mathtt{e} : {}^{\mathtt{s}}\mathtt{T} \\ \mathtt{h}, {}^{\mathtt{r}}\Gamma, \mathtt{e} \rightsquigarrow \mathtt{h}', \_ \end{array}\right\} \Rightarrow \left\{\begin{array}{l} \forall \iota \in dom(\mathtt{h}), \mathtt{f} : \\ \quad \mathtt{h}(\iota)\downarrow_2(\mathtt{f}) = \mathtt{h}'(\iota)\downarrow_2(\mathtt{f}) \ \vee \\ \quad owner(\mathtt{h}, {}^{\mathtt{r}}\Gamma(\mathtt{this})) \in owners(\mathtt{h}, \iota) \end{array}\right.$$

where $owner(\mathtt{h}, \iota)$ *denotes the direct owner of the object at address* ι *in heap* `h`*, and* $owners(\mathtt{h}, \iota)$ *denotes the set of all (transitive) owners of this object.*

The proof of Theorem 2 runs by rule induction on the operational semantics. The interesting cases are field update and calls of non-pure methods. In both cases, the type rules (Fig. 10) enforce that the receiver expression does not have the main modifier `any`. That is, the receiver object is owned by `this` or the owner of `this`. For the proof we assume that pure methods do not modify objects that exist in the prestate of the call. In this paper we do not describe how this is enforced in the program. A simple but conservative approach forbids all object creations, field updates, and calls of non-pure methods [20]. The above definition also allows weaker forms of purity that permit object creations [12] and also approaches that allow the modification of newly created objects [25].

6 Conclusion

We presented Generic Universe Types, an ownership type system for Java-like languages with generic types. Our type system permits arbitrary references through `any` types, but controls modifications of objects, that is, enforces the owner-as-modifier discipline. This allows us to handle interesting implementations beyond simple aggregate objects, for instance, shared buffers [12]. We show how `any` types and generics can be combined in a type safe way using limited covariance and viewpoint adaptation.

Generic Universe Types require little annotation overhead for programmers. As we have shown for non-generic Universe Types [12], this overhead can be further reduced by appropriate defaults. The default ownership modifier is generally `peer`, but the modifier of upper bounds, exceptions, and immutable types (such as `String`) defaults to `any`. These defaults make the conversion from Java 5 to Generic Universe Types simple.

The type checker and runtime support for Generic Universe Types are implemented in the JML tool suite [17].

As future work, we plan to use Generic Universe Types for program verification, extending our earlier work [20,21]. We are also working on path-dependent Universe Types to support more fine-grained information about object ownership, and to extend our inference tools for non-generic Universe Types to Generic Universe Types.

Acknowledgments. We are grateful to David Cunningham and to the anonymous ECOOP '07 and FOOL/WOOD '07 reviewers for their helpful comments. This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project, and the EPSRC grant Practical Ownership Types for Objects and Aspect Programs, EP/D061644/1.

References

1. Aldrich, J., Chambers, C.: Ownership domains: Separating aliasing policy from mechanism. In: Odersky, M. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 1–25. Springer, Heidelberg (2004)
2. Andrea, C., Coady, Y., Gibbs, C., Noble, J., Vitek, J., Zhao, T.: Scoped types and aspects for real-time systems. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 124–147. Springer, Heidelberg (2006)
3. Banerjee, A., Naumann, D.: Representation independence, confinement, and access control. In: Principles of Programming Languages (POPL), pp. 166–177. ACM Press, New York (2002)
4. Boyapati, C.: SafeJava: A Unified Type System for Safe Programming. PhD thesis, MIT (2004)
5. Boyapati, C., Lee, R., Rinard, M.: Ownership types for safe programming: Preventing data races and deadlocks. In: Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pp. 211–230. ACM Press, New York (2002)

6. Boyapati, C., Salcianu Jr., A., Beebee, W., Rinard, M.: Ownership types for safe region-based memory management in real-time Java. In: Programming language design and implementation (PLDI), pp. 324–337. ACM Press, New York (2003)
7. Clarke, D.: Object Ownership and Containment. PhD thesis, University of New South Wales (2001)
8. Clarke, D., Drossopoulou, S.: Ownership, encapsulation and the disjointness of type and effect. In: Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pp. 292–310. ACM Press, New York (2002)
9. Clarke, D.G., Potter, J.M., Noble, J.: Ownership types for flexible alias protection ACM SIGPLAN Notices. In: Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), vol. 33(10) (1998)
10. Dietl, W., Drossopoulou, S., Müller, P.: Formalization of Generic Universe Types. Technical Report 532, ETH Zurich (2006), `sct.inf.ethz.ch/publications`
11. Dietl, W., Müller, P.: Exceptions in ownership type systems. In: Poll, E. (ed.) Formal Techniques for Java-like Programs, pp. 49–54 (2004)
12. Dietl, W., Müller, P.: Universes: Lightweight ownership for JML. Journal of Object Technology (JOT) 4(8) (2005)
13. Emir, B., Kennedy, A.J., Russo, C., Yu, D.: Variance and generalized constraints for C# generics. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 279–303. Springer, Heidelberg (2006)
14. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: a minimal core calculus for Java and GJ. ACM Transactions on Programming Languages and Systems (TOPLAS) 23(3), 396–450 (2001)
15. Jacobs, B., Piessens, F., Leino, K.R.M., Schulte, W.: Safe concurrency for aggregate objects with invariants. In: Software Engineering and Formal Methods (SEFM), pp. 137–147. IEEE Computer Society Press, Los Alamitos (2005)
16. Kennedy, A., Syme, D.: Design and Implementation of Generics for the.NET Common Language Runtime. In: Programming Language Design and Implementation (PLDI), pp. 1–12 (2001)
17. Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J.: JML reference manual. Department of Computer Science, Iowa State University (2006), Available from `www.jmlspecs.org`
18. Leino, K.R.M., Müller, P.: Object invariants in dynamic contexts. In: Odersky, M. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 491–516. Springer, Heidelberg (2004)
19. Lu, Y., Potter, J.: Protecting representation with effect encapsulation. In: Principles of programming languages (POPL), pp. 359–371. ACM Press, New York (2006)
20. Müller, P. (ed.): Modular Specification and Verification of Object-Oriented Programs. LNCS, vol. 2262. Springer, Heidelberg (2002)
21. Müller, P., Poetzsch-Heffter, A., Leavens, G.T.: Modular invariants for layered object structures. Science of Computer Programming 62, 253–286 (2006)
22. Nägeli, S.: Ownership in design patterns. Master's thesis, ETH Zurich (2006), `sct.inf.ethz.ch/projects/student_docs/Stefan_Naegeli`
23. Noble, J., Vitek, J., Potter, J.M.: Flexible alias protection. In: Jul, E. (ed.) ECOOP 1998. LNCS, vol. 1445, Springer, Heidelberg (1998)
24. Potanin, A., Noble, J., Clarke, D., Biddle, R.: Generic ownership for generic Java. In: Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), October 2006, pp. 311–324. ACM Press, New York (2006)
25. Salcianu, A., Rinard, M.C.: Purity and side effect analysis for Java programs. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 199–215. Springer, Heidelberg (2005)