Generic Universe Types

Werner Dietl
ETH Zurich
Werner.Dietl@inf.ethz.ch

Sophia Drossopoulou
Imperial College London
S.Drossopoulou@imperial.ac.uk

Peter Müller
ETH Zurich
Peter.Mueller@inf.ethz.ch

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.

Keywords Ownership, generics, aliasing, Universe types

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 [21, 23, 24], thread synchronization [6, 18], memory management [2, 8], 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 [11] and their descendants [7, 9, 10, 27] 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 is enforced by the Universe type system [14], in Spec#'s dynamic ownership model [21], and Effective Ownership Types [22]. 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 [14, 25]. 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 [5] 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) [27] 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, 7, 26] 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 [14]. 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`.

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 modi-

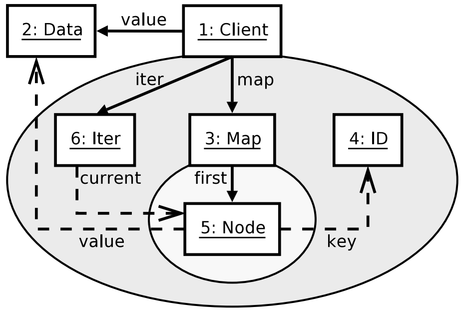

Figure 1. Object structure of a map from `ID` to `Data` objects. The map is represented by `Node` objects. The iterator has a direct reference to a node. Objects, references, and contexts are depicted by rectangles, arrows, and ellipses, respectively. Owner objects sit atop the context of objects they 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.

fier (analogous to the `readonly` modifier in non-generic Universe types [14]). 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 of this paper illustrates the main concepts of Generic Universe Types 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 [12].

2. Main Concepts

In this section, we explain the main concepts of Generic Universe Types 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 `Link` (both Fig. 3), which is used by the iterator class `Iter` (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. We chose an unusual iterator design to highlight several technical details of the type system. We will discuss this design and provide a more flexible solution at the end of this section.

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.

[1] We ignore static methods in this paper, but an extension is possible [23].

```
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;
  }

  V get(K key) {
    peer Iter<rep Node<K,V>> i = iterator();
    while (i.hasNext()) {
      rep Node<K,V> mn = i.next();
      if (mn.key.equals(key)) return mn.value;
    }
    return null;
  }

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

Figure 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`.

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.

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 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 the third parameter of `Node`'s method `init`. 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>`).

Type Parameters. Ownership modifiers are also used in actual type arguments. For instance, `Map`'s method `get` instantiates class `Iter` with the type argument `rep Node<K,V>`. Thus, local variable `i` has type `peer Iter<rep Node<K,V>>`, which has two ownership modifiers. The *main modifier* `peer`

```
class Link<X> {
  X next;
  void initLink(X n) { next = n; }
}

class Node<K,V> extends Link<peer Node<K,V>> {
  K key; V value;

  void init(K k, V v, peer Node<K,V> n) {
    initLink(n); key = k; value = v;
  }
}
```

Figure 3. Nodes form the internal representation of maps. Class Link implements rudimentary nodes for singly-linked lists. Its subclass Node instantiates Link's type parameter to implement a list of nodes with the same owner. It also adds attributes to store the keys and values of the map.

```
class Iter<X extends any Link<X>> {
  X current;

  void init(X start) { setCurrent(start); }
  void setCurrent(X c) { current = c; }
  pure boolean hasNext() { return current != null; }

  X next() {
    X result = current;
    current = current.next;
    return result;
  }
}
```

Figure 4. Class Iter implements iterators over Link structures. The precise node type is passed as type parameter. The upper bound allows method next to access a node's next field.

expresses that the Iter 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, i).

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 Iter<rep Node <K,V>> in class Map, we adapt the upper bound of Iter's type variable (any Link<X>) from viewpoint peer Iter<rep Node <K,V>> to the viewpoint this. With the appropriate substitutions, this adaptation yields any Link<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 initially instantiated. Therefore, the viewpoint is the same as for the instantiation, and no viewpoint adaptation is required. For instance, the call expression i.next() in method get (Fig. 2) has type rep Node<K,V>, because the result type of next() is the type variable X, which gets substituted by the type argument of i's type, rep Node<K,V>.

Therefore, even though an Iter object does not know the owner of the nodes it references (due to the any upper bound), clients of the iterator can recover the exact ownership information from the type argument. This illustrates that Generic Universe Types provide strong static guarantees similar to those of owner-parametric systems [11], 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 with the corresponding runtime check [14].

Limited Covariance and Viewpoint Adaptation of Type Arguments. Subtyping with covariant type arguments is in general not type safe. For instance, if List<String> was 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 Iter<rep Node<K,V>> was a subtype of peer Iter<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 [15].

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 Iter<rep Node<K,V>> is an admissible subtype of any Iter<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 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. As illustrated by Fig. 1, the most precise type for the call expression map.iterator() would be rep Iter<any Node<rep ID, any Data>> because the Iter 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 iterator'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 Iter<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.

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

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

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

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

```
class Pair<X,Y> {
  X x; Y y;
  void init(X px, Y py) { x = px; y = py; }
}

class PairIter<K,V> {
  any Node<K,V> current;

  void init(any Node<K,V> start) { current = start; }
  boolean hasNext() { return current != null; }

  peer Pair<K,V> next() {
    peer Pair<K,V> result = new peer Pair<K,V>();
    result.init(current.key, current.value);
    current = current.next;
    return result;
  }
}
```

Figure 6. A more sensible iterator implementation yielding key-value pairs.

Unfortunately, since method `next` is also non-pure, `main` must not call `iter.next()` either, which renders `Iter` objects useless outside the associated `Map` object. However, this is not a severe restriction since an iterator that exposes internal nodes should not be available to clients in the first place. That is, `Map`'s `iterator` method should be private. Alternatively, the iterator could yield pairs of keys and values rather than internal nodes. Such an iterator is shown in Fig. 6. The adapted implementation of the `iterator` method looks as follows:

```
  peer PairIter<K,V> iterator() {
    return new peer PairIter<K,V>(first);
  }
```

Since the type arguments of `iterator`'s result type are type variables, they are not subject to viewpoint adaptation. With this implementation of `iterator`, the type of the call expression `map.iterator()` (Fig. 5) is determined by adapting the main modifier and by performing the appropriate substitutions, which yields `rep PairIter<rep ID, any Data>`. Since the main modifier is `rep` rather than `any`, this type allows method `main` to call `next` on the iterator.

3. Static Checking

In this section, we formalize the compile time aspects of Generic Universe Types. 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 [5, 13, 23]). The language we use is similar to Featherweight Generic Java [17]. We added field updates because the treatment of side effects is essential for ownership type systems and especially the owner-as-modifier discipline.

Fig. 7 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).

We use the superscript ^{s} to distinguish the sorts for static checking from corresponding sorts that will be used to describe the runtime behavior. Whenever it is clear from context whether we refer to static or runtime entities, we omit the superscript ^{s}.

$\overline{\mathtt{T}}$ 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 (sort `Program`) consists of a sequence of classes, the identifier of a main class `C`, and a main expression `e`. 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 (`Class`) has a class identifier (`ClassId`), 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. Like in Java, each class directly or transitively extends the predefined class `Object`.

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

The sort `OM` of ownership modifiers contains $\mathtt{peer}_u$, $\mathtt{rep}_u$, and $\mathtt{any}_u$ 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 internally by the type system to distinguish accesses through `this` from other accesses, which simplifies the type rules. We omit the subscript u if it is clear from context that we mean an ownership modifier.

A method (`Meth`) consists of a signature and an expression as body. The result of evaluating the expression is returned by the method. The signature of a method (`MethSig`) consists of the method type variables with their upper bounds, the purity annotation, the return type, the method identifier (`MethId`), and the formal method parameters (ParId) with their types. Sort 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 [23].

The set of expressions (Expr) contains the null literal, method parameter access, field read, field update, method call, object creation, and cast.

Type checking is performed in a type environment (^{s}Env), which maps the type variables of the enclosing class and the enclosing method to their upper bounds and method parameters to their types. Since the domains of both mappings are disjoint, we use an overloaded notation and simply write $^s\Gamma(\mathtt{X})$ to refer to the upper bound of type variable X and $^s\Gamma(\mathtt{x})$ to refer to the type of method parameter x.

P	∈	Program	::=	$\overline{\texttt{Class}}$, ClassId, Expr
Cls	∈	Class	::=	class ClassId<$\overline{\texttt{TVarId}\ {}^s\texttt{NType}}$> extends ClassId<$\overline{{}^s\texttt{Type}}$> { $\overline{\texttt{FieldId}\ {}^s\texttt{Type};}$ $\overline{\texttt{Meth}}$ }
^{s}T	∈	^{s}Type	::=	^{s}NType | TVarId
^{s}N	∈	^{s}NType	::=	OM ClassId<$\overline{{}^s\texttt{Type}}$>
u	∈	OM	::=	$\texttt{peer}_u$ | $\texttt{rep}_u$ | $\texttt{any}_u$ | $\texttt{this}_u$
mt	∈	Meth	::=	MethSig { return Expr }
		MethSig	::=	<$\overline{\texttt{TVarId}\ {}^s\texttt{NType}}$> Purity ^{s}Type MethId($\overline{\texttt{ParId}\ {}^s\texttt{Type}}$)
w	∈	Purity	::=	pure | nonpure
e	∈	Expr	::=	null | ParId | Expr.FieldId | Expr.FieldId=Expr | Expr.MethId<$\overline{{}^s\texttt{Type}}$>($\overline{\texttt{Expr}}$) | new ^{s}Type | (^{s}Type) Expr
$^s\Gamma$	∈	^{s}Env	::=	$\overline{\texttt{TVarId}\ {}^s\texttt{NType};}$ $\overline{\texttt{ParId}\ {}^s\texttt{Type}}$

Figure 7. 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 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 ▷ 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 $\mathtt{e}_1$.f. Analogous adaptations occur for method parameters and results as well as upper bounds of type parameters. Let u be the main modifier of $\mathtt{e}_1$'s type, which expresses ownership relative to this. Let 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 $\mathtt{e}_1$.f has main modifier u▷u′.

$$\cdot \rhd \cdot \;::\; \mathtt{OM} \times \mathtt{OM} \rightarrow \mathtt{OM}$$
$$\begin{array}{lcllcll} \mathtt{this}\rhd\mathtt{u'} & = & \mathtt{u'} & \mathtt{rep}\rhd\mathtt{peer} & = & \mathtt{rep} & \\ \mathtt{peer}\rhd\mathtt{peer} & = & \mathtt{peer} & \mathtt{u}\rhd\mathtt{u'} & = & \mathtt{any} & \text{otherwise} \end{array}$$

The field access $\mathtt{e}_1$.f illustrates the motivation for this definition: (1) Accesses through this (that is, $\mathtt{e}_1$ is the variable this) do not require a viewpoint adaptation since the ownership modifier of the field is already relative to this. (2) If the main modifiers of both $\mathtt{e}_1$ and f are peer, then the object referenced by $\mathtt{e}_1$ has the same owner as this and the object referenced by $\mathtt{e}_1$.f has the same owner as $\mathtt{e}_1$ and, thus, the same owner as this. Consequently, the main modifier of $\mathtt{e}_1$.f is also peer. (3) If the main modifier of $\mathtt{e}_1$ is rep and the main modifier of f is peer, then the main modifier of $\mathtt{e}_1$.f is rep, because the object referenced by $\mathtt{e}_1$ is owned by this and the object referenced by $\mathtt{e}_1$.f has the same owner as $\mathtt{e}_1$, that is, this. (4) In all other cases, we cannot determine statically that the object referenced by $\mathtt{e}_1$.f has the same owner as this or is owned by this. Therefore, in these cases the main modifier of $\mathtt{e}_1$.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 $\rhd_m$ below and adapt the type arguments recursively:

$$\cdot \rhd \cdot \;::\; \mathtt{OM} \times {}^s\mathtt{Type} \rightarrow {}^s\mathtt{Type}$$
$$\begin{array}{lcl} \mathtt{u}\rhd\mathtt{X} & = & \mathtt{X} \\ \mathtt{u}\rhd\mathtt{N} & = & (\mathtt{u}\rhd_m\mathtt{N})\ \mathtt{C}\text{<}\overline{\mathtt{u}\rhd\mathtt{T}}\text{>} \text{ where } \mathtt{N} = \mathtt{u'}\ \mathtt{C}\text{<}\overline{\mathtt{T}}\text{>} \end{array}$$

The adapted main modifier is determined by u ▷ 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 N's type arguments contains the modifier rep, u′ is peer, and u is rep or peer. This leads to the following definition:

$$\cdot \rhd_m \cdot \;::\; \mathtt{OM} \times {}^s\mathtt{NType} \rightarrow \mathtt{OM}$$
$$\mathtt{u}\rhd_m\mathtt{u'}\ \mathtt{C}\text{<}\overline{\mathtt{T}}\text{>} \;=\; \begin{cases} \mathtt{any} & \text{if } (\mathtt{u} = \mathtt{rep} \vee \mathtt{u} = \mathtt{peer}) \wedge \\ & \quad \mathtt{u'} = \mathtt{peer} \wedge \mathtt{rep} \in \overline{\mathtt{T}} \\ \mathtt{u}\rhd\mathtt{u'} & \text{otherwise} \end{cases}$$

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 u.

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

$$\cdot \rhd \cdot \;::\; {}^s\mathtt{NType} \times {}^s\mathtt{Type} \rightarrow {}^s\mathtt{Type}$$
$$\mathtt{u}\ \mathtt{C}\text{<}\overline{\mathtt{T}}\text{>}\rhd\mathtt{T} \;=\; (\mathtt{u}\rhd\mathtt{T})[\overline{\mathtt{T}/\mathtt{X}}] \qquad \text{where } \overline{\mathtt{X}} = \mathrm{dom}(\mathtt{C}).$$

The operator ▷ adapts the ownership modifiers of T and then substitutes the type arguments $\overline{\mathtt{T}}$ for the type variables $\overline{\mathtt{X}}$ of 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 T's ownership modifiers. For a declaration class C<$\overline{\mathtt{X}\ \_}$> ..., dom(C) denotes C's type variables $\overline{\mathtt{X}}$.

Note that the first parameter is a non-variable type, because concrete ownership information 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 ▷.

Example. The call map.iterator() in method 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 Iter<rep Node<K,V>> (the return type of iterator) from rep Map<rep ID,any Data> (the type of the receiver expression). According to the above definition,

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

$$\text{SC-2}\ \frac{}{\texttt{C<}\overline{\texttt{T}}\texttt{>} \sqsubseteq \texttt{C<}\overline{\texttt{T}}\texttt{>}} \qquad \text{SC-3}\ \frac{\texttt{C<}\overline{\texttt{T}}\texttt{>} \sqsubseteq \texttt{C}''\texttt{<}\overline{\texttt{T}''}\texttt{>} \quad \texttt{C}''\texttt{<}\overline{\texttt{T}''}\texttt{>} \sqsubseteq \texttt{C}'\texttt{<}\overline{\texttt{T}'}\texttt{>}}{\texttt{C<}\overline{\texttt{T}}\texttt{>} \sqsubseteq \texttt{C}'\texttt{<}\overline{\texttt{T}'}\texttt{>}}$$

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

Figure 8. Rules for subclassing.

we first apply viewpoint adaption to the main modifier of the receiver type, `rep`, and the return type, and then substitute type variables.

The type arguments of the adapted type are obtained by applying viewpoint adaptation recursively to the type argument: `rep` $\triangleright$ `rep Node<K,V>`. This yields `any Node<K,V>` because `rep` $\triangleright$ `rep` $=$ `any`, and 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 Iter<rep Node<K,V>>`, which yields `any`.

So far, the adaptation yields `any Iter<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 `any Iter<any Node<rep ID,any Data>>` as the type of the call.

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 $\texttt{C<}\overline{\texttt{T}}\texttt{>}$. The subclass relation is the smallest relation satisfying the rules in Fig. 8. Each un-instantiated class is a subclass of the class it extends (SC-1). The form $\texttt{class C<}\overline{\texttt{X N}}\texttt{> extends C}'\texttt{<}\overline{\texttt{T}'}\texttt{>\{ }\overline{\texttt{f T}}\texttt{; }\overline{\texttt{m}}\texttt{ \}}$, 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. 10.

We illustrate subclassing by the classes `Link` and `Node` in Fig. 3. By rule SC-1, and from the `extends` clause of `Node`, we obtain `Node<K,V>` $\sqsubseteq$ `Link<peer Node<K,V>>`. Rule SC-4 allows us to instantiate the type variables, for instance, with the type arguments used in method `main`: `Node<rep ID,any Data>` $\sqsubseteq$ `Link<peer Node<rep ID, any Data>>`.

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 $\texttt{T}'$ in type environment Γ. The environment is needed since types may contain type variables. The rules for this subtyping judgment are presented in Fig. 9. Two types with the same main modifier are subtypes if the corresponding classes are subclasses (ST-1). 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`. Since both $\texttt{this}_u$ and `peer` 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 $<:_a$. 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` $<:_a$ `any Object`. Reflexivity of $<:$ follows from TA-1 and ST-5.

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

$$\text{ST-2}\ \frac{}{\Gamma \vdash \texttt{this}_u\ \texttt{C<}\overline{\texttt{T}}\texttt{>} <: \texttt{peer C<}\overline{\texttt{T}}\texttt{>}}$$

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

$$\text{ST-5}\ \frac{\texttt{T} <:_a \texttt{T}'}{\Gamma \vdash \texttt{T} <: \texttt{T}'}$$

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

Figure 9. Rules for subtyping and limited covariance.

In our example, using TA-1 twice (for `K` and `V`) and TA-2 we obtain `rep Node<K,V>` $<:_a$ `any Node<K,V>`. By TA-2 and ST-5 we derive `peer Iter<rep Node<K,V>>` $<:$ `any Iter<any Node<K,V>>`; an example for limited covariance. We cannot derive `peer Iter<rep Node<K,V>>` $<:$ `peer Iter<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 ${}^s\!fType(\texttt{C},\texttt{f})$ 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.

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

Method Lookup. The function $mType(\texttt{C},\texttt{m})$ 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.

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

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. 10 and explained in the following.

Well-Formed Types. The judgment $\Gamma \vdash$ 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 u C<$\overline{\mathtt{T}}$> is well-formed if its type arguments $\overline{\mathtt{T}}$ are well-formed and for each type parameter the actual type argument is a subtype of the upper bound, adapted from the viewpoint u C<$\overline{\mathtt{T}}$> (WFT-2). The viewpoint adaptation is necessary because the type arguments describe ownership relative to the this object where u C<$\overline{\mathtt{T}}$> is used, whereas the upper bounds are relative to the object of type u C<$\overline{\mathtt{T}}$>.

Note that rule WFT-2 permits type variables of a class C to be used in upper bounds of C. For instance in class Iter (Fig. 4), type variable X is used in its own upper bound, any Link<X>. To illustrate rule WFT-2, we show that the instantiation of class Iter in method main, any Iter<any Node<rep ID,any Data>> is well-formed.

The types rep ID and any Data are trivially well-formed. To show that the type any Node<rep ID,any Data> is well-formed, we have to show that the type arguments are subtypes of the adapted upper bounds (WFT-2). The (unadapted and adapted) upper bounds are any Object, which is a supertype of rep ID and any Data.

Finally, we have to show that any Node<rep ID,any Data> is a subtype of the adaptation of the upper bound any Link<X>. As illustrated in Sec. 3.2, we first perform the adaptation of ownership modifiers, which yields any Link<X>. Then we substitute the type variable X by the actual type argument, which yields any Link<any Node<rep ID,any Data>>. It is this substitution that makes it possible to use X in its own upper bound. Showing that any Node<rep ID,any Data> is a subtype of any Link<peer Node<rep ID,any Data>> is easy (ST-1, see Sec. 3.3); the latter is a subtype of any Link<any Node<rep ID,any Data>> (TA-1, TA-2, ST-5). By transitivity (ST-3), we have the desired relation.

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

Well-Formed Type Environments. The judgment Γ 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 this$_u$ and an uninstantiated class.

3.6 Type Rules

We are now ready to present the type rules (Fig. 11). The judgment $\Gamma \vdash$ e : 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. We also omit checks for valid appearances of the ownership modifier this$_u$. As explained earlier, this$_u$ must not occur in the program and is only used as main modifier of the type of this.

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 (GT-Null). The rules for object creation (GT-New) and cast (GT-Cast) are straightforward. GT-Cast 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 C$_0$. (Recall that *fType*(C$_0$, 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 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*(u$_0$, T$_1$) enforces that f is updated through receiver this if its declared type contains a rep modifier. In that case,

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

$$\text{WFM-1}\;\frac{\begin{array}{c}\Gamma = \overline{X_m\ N_b}, \overline{X\ N}; \mathtt{this}\ (\mathtt{this}_u\ C\text{<}\overline{X}\text{>}), \overline{x\ T_p} \\ \Gamma \vdash T_r, \overline{N_b}, \overline{T_p}\ \mathtt{ok} \qquad \Gamma \vdash e : T_r \\ \mathit{override}(C, m) \\ w = \mathtt{pure}\ \Rightarrow\ \overline{T_p} = \overline{\mathtt{any} \triangleright T_p}\end{array}}{\text{<}\overline{X_m\ N_b}\text{>}\ w\ T_r\ m(\overline{x\ T_p})\ \{\ \mathtt{return}\ e\ \}\ \mathtt{ok\ in}\ C\text{<}\overline{X\ N}\text{>}}$$

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

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

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

Figure 10. Well-formedness rules.

$$\text{GT-Subs}\;\frac{\begin{array}{c}\Gamma \vdash e : T \\ \Gamma \vdash T <: T'\end{array}}{\Gamma \vdash e : T'} \qquad \text{GT-Var}\;\frac{x \in \mathrm{dom}(\Gamma)}{\Gamma \vdash x : \Gamma(x)}$$

$$\text{GT-Null}\;\frac{}{\Gamma \vdash \mathtt{null} : T}$$

$$\text{GT-New}\;\frac{}{\Gamma \vdash \mathtt{new}\ T : T} \qquad \text{GT-Cast}\;\frac{\Gamma \vdash e_0 : T_0}{\Gamma \vdash (T)\ e_0 : T}$$

$$\text{GT-Read}\;\frac{\Gamma \vdash e_0 : N_0 \qquad N_0 = \_\ C_0\text{<}\_\text{>}}{\Gamma \vdash e_0.f : N_0 \triangleright \mathit{fType}(C_0, f)}$$

$$\text{GT-Upd}\;\frac{\begin{array}{c}\Gamma \vdash e_0 : N_0 \qquad N_0 = u_0\ C_0\text{<}\_\text{>} \\ T_1 = \mathit{fType}(C_0, f) \\ \Gamma \vdash e_2 : N_0 \triangleright T_1 \\ u_0 \neq \mathtt{any} \qquad \mathit{rp}(u_0, T_1)\end{array}}{\Gamma \vdash e_0.f{=}e_2 : N_0 \triangleright T_1}$$

$$\text{GT-Invk}\;\frac{\begin{array}{c}\Gamma \vdash e_0 : N_0 \qquad N_0 = u_0\ C_0\text{<}\_\text{>} \\ \mathit{mType}(C_0, m) = \text{<}\overline{X_m\ N_b}\text{>}\ w\ T_r\ m(\overline{x\ T_p}) \\ \Gamma \vdash \overline{T} <: (N_0 \triangleright \overline{N_b})[\overline{T/X_m}] \qquad \Gamma \vdash \overline{e_2} : (N_0 \triangleright \overline{T_p})[\overline{T/X_m}] \\ (u_0 = \mathtt{any} \Rightarrow w = \mathtt{pure}) \qquad \mathit{rp}(u_0, \overline{T_p} \circ \overline{N_b})\end{array}}{\Gamma \vdash e_0.m\text{<}\overline{T}\text{>}(\overline{e_2}) : (N_0 \triangleright T_r)[\overline{T/X_m}]}$$

Figure 11. Type rules.

the viewpoint adaptation $N_0 \triangleright T_1$ 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`:

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

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 N_0 and subsumption. The type of the invocation expression is determined by viewpoint adaptation of the return type T_r from the receiver type N_0. Modulo subsumption, the actual method arguments must have the formal parameter types, adapted from N_0 and with actual type arguments $\overline{T}$ substituted for the method's type variables 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>` ▷ `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`. This requirement prevents method `main` (Fig. 5) from calling `iter.next()` as discussed in Sec. 2. 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. 10). 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. 12 defines our model of the heap. The prefix ^{r} distinguishes sorts of the runtime model from their static counterparts.

h	$\in$	Heap	=	Addr → Obj
ι	$\in$	Addr	=	Set of Addresses ∪ $\{\mathtt{null}_a\}$
o	$\in$	Obj	=	^{r}Type, Fields
^{r}T	$\in$	^{r}Type	=	OwnerAddr ClassId<$\overline{{}^r\text{Type}}$>
Fs	$\in$	Fields	=	FieldId → Addr
ι	$\in$	OwnerAddr	=	Addr ∪ $\{\mathtt{any}_a\}$
${}^r\Gamma$	$\in$	^{r}Env	=	$\overline{\text{TVarId}\ {}^r\text{Type};}\ \overline{\text{ParId Addr}}$

Figure 12. Definitions for the heap model.

A heap (sort `Heap`) maps addresses to objects. The set of addresses (`Addr`) contains the special null-reference $\mathtt{null}_a$.

An object (Obj) consist of its runtime type and a mapping from field identifiers to the addresses stored in the fields.

The runtime type (^{r}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 [19]. 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 ι. The runtime type of the object stored in map is ι Map<ι ID, $\mathtt{any}_a$ Data>.

The first component of a runtime environment (^{r}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 $\iota \vdash {}^{r}T \; {}^{r}{<:} \; {}^{r}T'$ expresses that the runtime type ^{r}T is a subtype of ^{r}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. 13. According to RT-1, subtyping follows subclassing if (1) the runtime types have the same owner address, (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 $\mathtt{any}_a$, and (5) the type variables $\overline{\mathtt{X}}$ of the subclass C are substituted consistently by $\overline{^{r}\mathtt{T}}$. Note that in a well-formed program, $\mathtt{this}_u$ never occurs in a type argument; nevertheless we include the substitution for consistency. According to RT-2, subtyping is transitive.

As for subtyping for static types, we have limited covariance for runtime types. Covariant subtyping is expressed by the relation $^{r}{<:}_{a}$. Two runtime types are subtypes if they are covariant subtypes (RT-3). The rules for limited covariance, RTA-1 and RTA-2, are analogous to the rules TA-1 and TA-2 for static types (Fig. 9). Reflexivity of $^{r}{<:}$ follows from RTA-1 and RT-3.

In Sec. 3.3, we got Node<K, V> $\sqsubseteq$ Link<peer Node<K, V>>. By RT-1, we get the runtime relation: $\iota \vdash \iota'$ Node<$^{r}T_1$, $^{r}T_2$> $^{r}{<:}$ ι' Link<ι' Node<$^{r}T_1$, $^{r}T_2$>>.

The judgment $\mathtt{h} \vdash \iota : {}^{r}T'$ expresses that in heap h, the address ι has type ^{r}T′. The type of ι is determined by the type of the object at ι and the subtype relation (RT-4). The null reference can have any type (RT-5).

From Static Types to Runtime Types. Static types and runtime types are related by the following *dynamization function*, which is defined by rule DYN (Fig. 14):

$$dyn :: {}^{s}\mathtt{Type} \times \mathtt{Addr} \times {}^{r}\mathtt{Type} \times {}^{r}\mathtt{Env} \rightarrow {}^{r}\mathtt{Type}$$

This function maps a static type ^{s}T to the corresponding runtime type. The viewpoint is described by an address ι and a runtime type ^{r}T (usually the address and runtime type of the this object). In ^{s}T, *dyn* substitutes rep by ι, peer and $\mathtt{this}_u$ by the owner in ^{r}T, ι', and $\mathtt{any}_u$ by $\mathtt{any}_a$. It also substitutes all type variables in ^{s}T by the instantiations given in ι' C<$\overline{^{r}\mathtt{T}}$>, a supertype of ι's runtime type, or in the runtime environment. The substitutions performed by *dyn* are analogous to the ones in rule RT-1 (Fig. 13), which also involves mapping static types to runtime types. We do not use *dyn* in RT-1 to avoid that the definitions of $^{r}{<:}$ and *dyn* are mutually recursive.

Note that the outcome of *dyn* depends on finding ι' C<$\overline{^{r}\mathtt{T}}$>, an appropriate supertype of ^{r}T, which contains substitutions for all type variables not mapped by the environment (free(^{s}T) yields the free type variables in ^{s}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 ^{s}T are in the domain of a class then they are not in the domain of any other class. In addition, we want the most concrete runtime type arguments. To ensure this, we require that all other supertypes of ^{r}T that have the same class C have less specific type arguments.

To illustrate dynamization, consider an execution of method put (Fig. 2), whose this object has address ι and runtime type ι' Map<$^{r}T_1$, $^{r}T_2$>. Now 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. this is dyn(rep Node<K, V>, ι, ι' Map<$^{r}T_1$, $^{r}T_2$>, _), which yields ι Node<$^{r}T_1$, $^{r}T_2$>. This runtime type correctly reflects that the new object is owned by this (owner address ι) and has the same type arguments as the runtime type of this.

It is convenient to define the following three overloaded versions of *dyn*. 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.

$$\begin{array}{lcl} dyn({}^{s}T, \iota, {}^{r}T) & = & dyn({}^{s}T, \iota, {}^{r}T, \epsilon) \\ dyn({}^{s}T, \iota, \mathtt{h}, {}^{r}\Gamma) & = & dyn({}^{s}T, \iota, \mathtt{h}(\iota)\downarrow_1, {}^{r}\Gamma) \\ dyn({}^{s}T, \mathtt{h}, {}^{r}\Gamma) & = & dyn({}^{s}T, {}^{r}\Gamma(\mathtt{this}), \mathtt{h}, {}^{r}\Gamma) \end{array}$$

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 $^{r}fType(\iota, {}^{r}T, \mathtt{f})$ yields the runtime type of f in an object at address ι with runtime type ^{r}T. In the definition of $^{r}fType$ (rule RFT in Fig. 14), C′ is the superclass of the class C of ^{r}T, in which f is actually defined.

Method Lookup. The function $mBody(\mathtt{C}, \mathtt{m})$ 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. 14). Otherwise, m is looked up in C's superclass 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. 14.

Well-Formed Runtime Types. The judgment $\iota \vdash \iota'$ C<$\overline{^{r}\mathtt{T}}$> ok expresses that runtime type ι' C<$\overline{^{r}\mathtt{T}}$> is well-formed for viewpoint address ι. According to rule WFRT, 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. Since upper bounds of well-formed classes do not contain rep (see rule WFC in Fig. 10), we can pass an arbitrary address to *dyn*.

Well-Formed Heaps. A heap h is well-formed, denoted by h ok, if and only if the runtime types of all objects are well-formed, the $\mathtt{null}_a$ address is not mapped to an object, and all addresses stored in fields are well-typed (WFH).

$$\text{RT-1}\ \frac{\mathtt{C}\texttt{<}\overline{\mathtt{X}}\texttt{>} \sqsubseteq \mathtt{C'}\texttt{<}\overline{{}^{s}\mathtt{T}}\texttt{>} \qquad \mathrm{dom}(\mathtt{C}) = \overline{\mathtt{X}}}{\iota \vdash \iota'\ \mathtt{C}\texttt{<}\overline{{}^{r}\mathtt{T}}\texttt{>}\ {}^{r}\!\!<:\ \iota'\ \mathtt{C'}\texttt{<}\overline{{}^{s}\mathtt{T}}[\iota'/\mathtt{this}_u, \iota'/\mathtt{peer}, \iota/\mathtt{rep}, \mathtt{any}_a/\mathtt{any}_u, \overline{{}^{r}\mathtt{T}/\mathtt{X}}]\texttt{>}} \qquad \text{RT-2}\ \frac{\iota \vdash {}^{r}\mathtt{T}\ {}^{r}\!\!<:\ {}^{r}\mathtt{T''} \quad \iota \vdash {}^{r}\mathtt{T''}\ {}^{r}\!\!<:\ {}^{r}\mathtt{T'}}{\iota \vdash {}^{r}\mathtt{T}\ {}^{r}\!\!<:\ {}^{r}\mathtt{T'}} \qquad \text{RT-3}\ \frac{{}^{r}\mathtt{T}\ {}^{r}\!\!<:_{\mathtt{a}}\ {}^{r}\mathtt{T'}}{\iota \vdash {}^{r}\mathtt{T}\ {}^{r}\!\!<:\ {}^{r}\mathtt{T'}}$$

$$\text{RTA-1}\ \frac{}{{}^{r}\mathtt{T}\ {}^{r}\!\!<:_{\mathtt{a}}\ {}^{r}\mathtt{T}} \qquad \text{RTA-2}\ \frac{\overline{{}^{r}\mathtt{T}\ {}^{r}\!\!<:_{\mathtt{a}}\ {}^{r}\mathtt{T'}}}{\iota'\ \mathtt{C}\texttt{<}\overline{{}^{r}\mathtt{T}}\texttt{>}\ {}^{r}\!\!<:_{\mathtt{a}}\ \mathtt{any}_a\ \mathtt{C}\texttt{<}\overline{{}^{r}\mathtt{T'}}\texttt{>}} \qquad \text{RT-4}\ \frac{h(\iota) = {}^{r}\mathtt{T}, \_ \quad \iota \vdash {}^{r}\mathtt{T}\ {}^{r}\!\!<:\ {}^{r}\mathtt{T'}}{\mathtt{h} \vdash \iota : {}^{r}\mathtt{T'}} \qquad \text{RT-5}\ \frac{}{\mathtt{h} \vdash \mathtt{null}_a : {}^{r}\mathtt{T'}}$$

Figure 13. Rules for subtyping on runtime types.

$$\text{DYN}\ \frac{\begin{array}{c}{}^{r}\mathtt{T} = \iota'\ \_\texttt{<}\_\texttt{>} \quad \iota \vdash {}^{r}\mathtt{T}\ {}^{r}\!\!<:\ \iota'\ \mathtt{C}\texttt{<}\overline{{}^{r}\mathtt{T}}\texttt{>} \quad \iota \vdash {}^{r}\mathtt{T}\ {}^{r}\!\!<:\ \iota'\ \mathtt{C}\texttt{<}\overline{{}^{r}\mathtt{T}_a}\texttt{>} \Rightarrow \iota \vdash \overline{{}^{r}\mathtt{T}}\ {}^{r}\!\!<:\ \overline{{}^{r}\mathtt{T}_a} \\ \mathrm{dom}(\mathtt{C}) = \overline{\mathtt{X}} \qquad \mathrm{free}({}^{s}\mathtt{T}) \subseteq \overline{\mathtt{X}} \circ \overline{\mathtt{X'}}\end{array}}{dyn({}^{s}\mathtt{T}, \iota, {}^{r}\mathtt{T}, (\overline{\mathtt{X'}\ {}^{r}\mathtt{T'}};\_)) = {}^{s}\mathtt{T}[\iota'/\mathtt{this}, \iota'/\mathtt{peer}, \iota/\mathtt{rep}, \mathtt{any}_a/\mathtt{any}_u, \overline{{}^{r}\mathtt{T}/\mathtt{X}}, \overline{{}^{r}\mathtt{T'}/\mathtt{X'}}]}$$

$$\text{RFT}\ \frac{{}^{r}\mathtt{T} = \_\ \mathtt{C}\texttt{<}\_\texttt{>} \qquad \mathtt{C}\texttt{<}\_\texttt{>} \sqsubseteq \mathtt{C'}\texttt{<}\_\texttt{>}}{{}^{r}\!fType(\iota, {}^{r}\mathtt{T}, \mathtt{f}) = dyn({}^{s}\!fType(\mathtt{C'}, \mathtt{f}), \iota, {}^{r}\mathtt{T})}$$

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

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

$$\text{WFRT}\ \frac{\begin{array}{c}\mathtt{class\ C}\texttt{<}\overline{\mathtt{X}\ {}^{s}\mathtt{N}}\texttt{>}\ \ldots \\ \iota \vdash \overline{{}^{r}\mathtt{T}}\ {}^{r}\!\!<:\ dyn(\overline{{}^{s}\mathtt{N}}, \_, \iota'\ \mathtt{C}\texttt{<}\overline{{}^{r}\mathtt{T}}\texttt{>})\end{array}}{\iota \vdash \iota'\ \mathtt{C}\texttt{<}\overline{{}^{r}\mathtt{T}}\texttt{>}\ \mathtt{ok}} \qquad \text{WFH}\ \frac{\begin{array}{c}(\forall\iota:\ \ \iota \vdash \mathtt{h}(\iota)\downarrow_1\ \ \mathtt{ok}) \qquad \mathtt{null}_a \notin \mathrm{dom}(\mathtt{h}) \\ (\forall\iota, \mathtt{f}:\ \ \mathtt{h}(\iota) = {}^{r}\mathtt{T}, \mathtt{Fs}\ \wedge\ {}^{r}\!fType(\iota, {}^{r}\mathtt{T}, \mathtt{f}) = {}^{r}\mathtt{T'} \implies \mathtt{h} \vdash \mathtt{Fs}(\mathtt{f}) : {}^{r}\mathtt{T'})\end{array}}{\mathtt{h\ ok}}$$

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

Figure 14. Rules for dynamization of static types, field and method lookup, and well-formedness.

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

4.4 Operational Semantics

We describe program execution by a big-step operational semantics. The transition $\mathtt{h}, {}^{r}\Gamma, \mathtt{e} \leadsto \mathtt{h'}, \iota$ expresses that the evaluation of an expression `e` in heap `h` and runtime environment ${}^{r}\Gamma$ results in address ι and successor heap $\mathtt{h'}$. A program with main class `C` is executed by evaluating the main expression in a heap $\mathtt{h}_0$ that contains exactly one `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 ${}^{r}\Gamma_0$ that maps `this` to this `C` instance (${}^{r}\Gamma_0 = \epsilon;\mathtt{this}\ \iota$). The rules for evaluating expressions are presented in Fig. 15 and explained in the following.

Parameters, including `this`, are evaluated by looking up the stored address in the stack, which is part of the runtime environment ${}^{r}\Gamma$ (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 has the runtime type that is the dynamization of the static type given in the cast expression w.r.t. the current `this` object (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`. Runtime information about type arguments and ownership is mainly required to check casts. However, we also use this information to evaluate `new` expressions, where the type is a type variable.

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

Rule OS-Invk describes how to evaluate a call of a method `m`. We evaluate the receiver expression and the 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.

$$\text{OS-Var}\ \frac{}{\mathtt{h}, {}^{r}\Gamma, \mathtt{x} \rightsquigarrow \mathtt{h}, {}^{r}\Gamma(\mathtt{x})} \qquad \text{OS-Null}\ \frac{}{\mathtt{h}, {}^{r}\Gamma, \mathtt{null} \rightsquigarrow \mathtt{h}, \mathtt{null}_a} \qquad \text{OS-Cast}\ \frac{\mathtt{h}, {}^{r}\Gamma, \mathtt{e}_0 \rightsquigarrow \mathtt{h}', \iota \quad \mathtt{h}' \vdash \iota : dyn({}^{s}\mathtt{T}, \mathtt{h}, {}^{r}\Gamma)}{\mathtt{h}, {}^{r}\Gamma, ({}^{s}\mathtt{T})\ \mathtt{e}_0 \rightsquigarrow \mathtt{h}', \iota}$$

$$\text{OS-New}\ \frac{\begin{array}{c}\iota \notin \mathrm{dom}(h) \quad \iota \neq \mathtt{null}_a \\ {}^{r}\mathtt{T} = dyn({}^{s}\mathtt{T}, \mathtt{h}, {}^{r}\Gamma) \\ {}^{r}\mathtt{T} = \_\ \mathtt{C<\_>} \\ \mathtt{Fs}(\mathit{fields}(\mathtt{C})) = \mathtt{null}_a \\ \mathtt{h}' = \mathtt{h}[\iota \mapsto ({}^{r}\mathtt{T}, \mathtt{Fs})]\end{array}}{\mathtt{h}, {}^{r}\Gamma, \mathtt{new}\ {}^{s}\mathtt{T} \rightsquigarrow \mathtt{h}', \iota} \qquad \text{OS-Read}\ \frac{\begin{array}{c}\mathtt{h}, {}^{r}\Gamma, \mathtt{e}_0 \rightsquigarrow \mathtt{h}', \iota_0 \\ \iota_0 \neq \mathtt{null}_a \\ \iota = \mathtt{h}'(\iota_0)\downarrow_2 (\mathtt{f})\end{array}}{\mathtt{h}, {}^{r}\Gamma, \mathtt{e}_0.\mathtt{f} \rightsquigarrow \mathtt{h}', \iota} \qquad \text{OS-Upd}\ \frac{\begin{array}{c}\mathtt{h}, {}^{r}\Gamma, \mathtt{e}_0 \rightsquigarrow \mathtt{h}_0, \iota_0 \\ \iota_0 \neq \mathtt{null}_a \\ \mathtt{h}_0, {}^{r}\Gamma, \mathtt{e}_2 \rightsquigarrow \mathtt{h}_2, \iota \\ \mathtt{h}' = \mathtt{h}_2[\iota_0.\mathtt{f} := \iota]\end{array}}{\mathtt{h}, {}^{r}\Gamma, \mathtt{e}_0.\mathtt{f}\mathtt{=}\mathtt{e}_2 \rightsquigarrow \mathtt{h}', \iota}$$

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

Figure 15. Operational semantics.

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 well-formed runtime environment as well as two objects $\mathtt{o}_1$ and $\mathtt{o}_2$. If from the viewpoint `this`, $\mathtt{o}_1$ has the type that is the dynamization of a static type ${}^{s}\mathtt{N}$, and from viewpoint $\mathtt{o}_1$, $\mathtt{o}_2$ has the type that is the dynamization of a static type ${}^{s}\mathtt{T}$, then from the viewpoint `this`, $\mathtt{o}_2$ has the type that is the dynamization of ${}^{s}\mathtt{T}$ adapted from ${}^{s}\mathtt{N}$, ${}^{s}\mathtt{N} \rhd {}^{s}\mathtt{T}$. 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. For simplicity, we omit the substitutions of method type variables (but see [12]). The last two requirements ensure that `rep` is interpreted correctly and that the adapted type ${}^{s}\mathtt{N} \rhd {}^{s}\mathtt{T}$ is well-formed.

Lemma 5.1 (Adaptation from a Viewpoint).

$$\left.\begin{array}{l}\mathtt{h} \vdash {}^{r}\Gamma : {}^{s}\Gamma \\ \mathtt{h} \vdash \iota_1 : dyn({}^{s}\mathtt{N}, \mathtt{h}, {}^{r}\Gamma) \\ \mathtt{h} \vdash \iota_2 : dyn({}^{s}\mathtt{T}, \iota_1, \mathtt{h}(\iota_1)\downarrow_1) \\ {}^{s}\mathtt{N} = \mathtt{u}_N\ \mathtt{C}_N\mathtt{<\_>} \\ \mathtt{u}_N = \mathtt{this}_u \Rightarrow {}^{r}\Gamma(\mathtt{this}) = \iota_1 \\ \mathit{free}({}^{s}\mathtt{T}) \subseteq \mathit{dom}(\mathtt{C}_N)\end{array}\right\} \implies \mathtt{h} \vdash \iota_2 : dyn({}^{s}\mathtt{N} \rhd {}^{s}\mathtt{T}, \mathtt{h}, {}^{r}\Gamma)$$

This lemma justifies the type rule GT-Read. The proof runs by induction on the shape of static type ${}^{s}\mathtt{T}$. 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 ${}^{s}\mathtt{N}$ and ${}^{s}\mathtt{T}$.

The following lemma is the converse of Lemma 5.1. 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 type that is the dynamization of a static type ${}^{s}\mathtt{N}$ and $\mathtt{o}_2$ has the type that is the dynamization of ${}^{s}\mathtt{N} \rhd {}^{s}\mathtt{T}$, then from viewpoint $\mathtt{o}_1$, $\mathtt{o}_2$ has the type that is the dynamization of a static type ${}^{s}\mathtt{T}$. The lemma requires that the adaptation of ${}^{s}\mathtt{T}$ does not change ownership modifiers in ${}^{s}\mathtt{T}$ from non-`any` to `any`, because the lost ownership information cannot be recovered. Such a change occurs if ${}^{s}\mathtt{N}$'s main modifier is `any` or if ${}^{s}\mathtt{T}$ contains `rep` and is not accessed through `this` (see definition of *rp*, Sec. 3.6).

Lemma 5.2 (Adaptation to a Viewpoint).

$$\left.\begin{array}{l}\mathtt{h} \vdash {}^{r}\Gamma : {}^{s}\Gamma \\ \mathtt{h} \vdash \iota_1 : dyn({}^{s}\mathtt{N}, \mathtt{h}, {}^{r}\Gamma) \\ \mathtt{h} \vdash \iota_2 : dyn({}^{s}\mathtt{N} \rhd {}^{s}\mathtt{T}, \mathtt{h}, {}^{r}\Gamma) \\ {}^{s}\mathtt{N} = \mathtt{u}_N\ \mathtt{C}_N\mathtt{<\_>} \\ \mathtt{u}_N \neq \mathtt{any}, \quad rp(\mathtt{u}_N, {}^{s}\mathtt{T}) \\ \mathtt{u}_N = \mathtt{this}_u \Rightarrow {}^{r}\Gamma(\mathtt{this}) = \iota_1 \\ \mathit{free}({}^{s}\mathtt{T}) \subseteq \mathit{dom}(\mathtt{C}_N)\end{array}\right\} \implies \mathtt{h} \vdash \iota_2 : dyn({}^{s}\mathtt{T}, \iota_1, \mathtt{h}(\iota_1)\downarrow_1)$$

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 5.1.

Type Safety. Type safety of 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. Moreover, our theorem expresses that the main modifier $\mathtt{this}_u$ is used solely for the type of `this`.

Theorem 5.3 (Type Safety).

$$\left.\begin{array}{l}\mathtt{h} \vdash {}^{r}\Gamma : {}^{s}\Gamma \\ {}^{s}\Gamma \vdash \mathtt{e} : {}^{s}\mathtt{T} \\ \mathtt{h}, {}^{r}\Gamma, \mathtt{e} \rightsquigarrow \mathtt{h}', \iota\end{array}\right\} \implies \left\{\begin{array}{l}\mathtt{h}' \vdash {}^{r}\Gamma : {}^{s}\Gamma \\ \mathtt{h}' \vdash \iota : dyn({}^{s}\mathtt{T}, \mathtt{h}, {}^{r}\Gamma) \\ {}^{s}\mathtt{T} = \mathtt{this}_u\ \mathtt{\_<\_>} \Rightarrow \\ \qquad \iota \in \{{}^{r}\Gamma(\mathtt{this}), \mathtt{null}_a\}\end{array}\right.$$

The proof of Theorem 5.3 runs by rule induction on the operational semantics. Lemma 5.1 is used to prove field read and method results, whereas Lemma 5.2 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 [17] and extensions for field updates and casts [16, 4]. The new runtime ownership check in casts can be treated analogously to standard Java casts.

Owner-as-Modifier. A property that is relevant for Generic Universe Types is the enforcement of the owner-as-modifier discipline, which 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 5.4 (Owner-as-Modifier).

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

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

The proof of Theorem 5.4 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. 11) 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 [23]. The above definition also allows weaker forms of purity that permit object creations [14] and also approaches that allow the modification of newly created objects [28].

6. Conclusions

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 [14]. 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 [14], 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 non-generic Universe Types are implemented in JML [20]. An adaptation to Generic Universe Types is ongoing.

As future work, we plan to use Generic Universe Types for program verification, extending our earlier work [23, 24]. One of the interesting challenges there is to relax the restrictions on `any` types, for instance, to allow field updates and calls of non-pure methods on receivers whose type is a type variable with an `any` upper bound. We are working on Path-dependent Universe Types to support more fine-grained information about object ownership [26], and plan to extend our existing inference tools for non-generic Universe Types to Generic Universe Types.

Acknowledgments

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.

References

[1] J. Aldrich and C. Chambers. Ownership domains: Separating aliasing policy from mechanism. In M. Odersky, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *LNCS*, pages 1–25. Springer-Verlag, 2004.

[2] C. Andrea, Y. Coady, C. Gibbs, J. Noble, J. Vitek, and T. Zhao. Scoped types and aspects for real-time systems. In *European Conference on Object Oriented Programming (ECOOP)*, LNCS. Springer-Verlag, 2006. To appear.

[3] A. Banerjee and D. Naumann. Representation independence, confinement, and access control. In *Principles of Programming Languages (POPL)*, pages 166–177. ACM, 2002.

[4] G.M. Bierman, M.J. Parkinson, and A.M. Pitts. An imperative core calculus for Java and Java with effects. Technical Report 563, University of Cambridge Computer Laboratory, April 2003.

[5] C. Boyapati. *SafeJava: A Unified Type System for Safe Programming*. PhD thesis, MIT, 2004.

[6] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 211–230. ACM Press, 2002.

[7] C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *Principles of Programming Languages (POPL)*, pages 213–223. ACM Press, 2003.

[8] C. Boyapati, A. Salcianu, Jr. W. Beebee, and M. Rinard. Ownership types for safe region-based memory management in real-time Java. In *Programming language design and implementation (PLDI)*, pages 324–337. ACM Press, 2003.

[9] D. Clarke. *Object Ownership and Containment*. PhD thesis, University of New South Wales, 2001.

[10] D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 292–310. ACM Press, 2002.

[11] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 33(10) of *ACM SIGPLAN Notices*, 1998.

[12] W. Dietl, S. Drossopoulou, and P. Müller. Formalization of Generic Universe Types. Technical Report 532, ETH Zurich, 2006. `sct.inf.ethz.ch/publications`.

[13] W. Dietl and P. Müller. Exceptions in ownership type systems. In E. Poll, editor, *Formal Techniques for Java-like Programs*, pages 49–54, 2004.

[14] W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology (JOT)*, 4(8), 2005.

[15] B. Emir, A. J. Kennedy, C. Russo, and D. Yu. Variance and generalized constraints for C# generics. In *European Conference on Object Oriented Programming (ECOOP)*, LNCS. Springer-Verlag, 2006. To appear.

[16] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. A programmer's reduction semantics for classes and mixins. In *Formal Syntax and Semantics of Java*, volume 1523 of *LNCS*, pages 241–269. Springer-Verlag, 1999.

[17] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(3):396–450, 2001.

[18] B. Jacobs, F. Piessens, K. R. M. Leino, and W. Schulte. Safe concurrency for aggregate objects with invariants. In

Software Engineering and Formal Methods (SEFM), pages 137–147. IEEE Computer Society, 2005.

[19] A. Kennedy and D. Syme. Design and Implementation of Generics for the .NET Common Language Runtime. In *Programming Language Design and Implementation (PLDI)*, pages 1–12, 2001.

[20] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, and J. Kiniry. JML reference manual. Department of Computer Science, Iowa State University. Available from `www.jmlspecs.org`, 2006.

[21] K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In M. Odersky, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *LNCS*, pages 491–516. Springer-Verlag, 2004.

[22] Y. Lu and J. Potter. Protecting representation with effect encapsulation. In *Principles of programming languages (POPL)*, pages 359–371. ACM Press, 2006.

[23] P. Müller. *Modular Specification and Verification of Object-Oriented programs*, volume 2262 of *LNCS*. Springer-Verlag, 2002.

[24] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 62:253–286, 2006.

[25] S. Nägeli. Ownership in design patterns. Master's thesis, ETH Zurich, 2006. `sct.inf.ethz.ch/projects/student_docs/Stefan_Naegeli`.

[26] M. Odersky, V. Cremet, C. Röckl, and M. Zenger. A nominal theory of objects with dependent types. In L. Cardelli, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 2743 of *LNCS*, pages 201–224. Springer-Verlag, 2003.

[27] A. Potanin, J. Noble, D. Clarke, and R. Biddle. Generic ownership for generic Java. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, ACM SIGPLAN Notices. ACM, 2006. To appear.

[28] A. Salcianu and M. C. Rinard. Purity and side effect analysis for Java programs. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 3385 of *LNCS*, pages 199–215. Springer-Verlag, 2005.