Generic Universe Types in JML

Robin Züger

Master Project Report

Software Component Technology Group
Department of Computer Science
ETH Zurich

http://sct.inf.ethz.ch/

July 2007

Supervised by:
Dipl.-Ing. Werner M. Dietl
Prof. Dr. Peter Müller

Software Component Technology Group
inf | Informatik
Computer Science

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract

Ownership is a powerful concept to structure the object store and to control aliasing and modifications of objects. Generic Universe Types is the first type system to combine the owner-as-modifier discipline with type genericity.

This report presents the implementation of Generic Universe Types in the Java Modeling Language. It also describes how concepts of Java such as arrays, exceptions, raw types and wildcards can be added. Furthermore, an approach is presented to infer the ownership structure for a normal Java program with generic types.

Contents

Chapter 1

Introduction

1.1 Ownership

In object-oriented programs, we have very limited means to control access to objects. Basically, an object can reference any other object in the object store and modify it by calling a method on it, or directly by accessing its fields. Even though most languages provide access modifiers which can limit access to, for example, a certain class or package, we still cannot be sure if we have the only reference to an object. There might be another reference pointing to the same object and allowing to change the object. This is what we call *aliasing*.

Aliasing helps making programs more efficient because it allows to pass a reference instead of copying the entire object (*pass-by-reference* versus *pass-by-value*). The drawbacks are for example *leaking* and *capturing*. Leaking occurs when an internal reference, i.e. a private field, gets passed out, e.g. by a getter, which then allows to alter the referenced object. Capturing is the opposite: When an object is passed to another object, e.g. by a setter, the existing reference can still be used to do modifications to the object.

Since every reference may be used to modify the referenced object, object-oriented programs can be hard to understand, to maintain, and to reason about. Especially modular verification of functional properties typically requires control of how references are passed around and which operations can be performed on them.

Ownership is one approach to solve this problem. It structures the object store hierarchically. Each object either has zero or one *owner*. The owner is another object that has a certain control over its children, i.e. the objects it owns. A *context* is the set of all objects having the same owner. The objects having no owner are in the so-called *root context*. The contexts build a tree whose root is the root context.

Ownership is a very powerful concept that can be used to solve various problems, for instance, memory management, representation independence or program verification. Depending on the problem to be solved, the owner has a different role.

For memory management and representation independence, the owner needs to be able to control how the objects it (transitively) owns are accessed. This is what the *owner-as-dominator* property is used for. It requires that all reference chains from an object in the root context to an object o in a different context go through o's owner.

For program verification, e.g. to prove correctness of invariants, a weaker ownership model suffices. The *owner-as-modifier* property allows any object to reference an object o, but reference chains that do not pass through o's owner must not be used to modify o. This makes it possible for the owner to control modifications to owned objects and therefore maintain invariants, but it cannot control who reads data. This is done is by introducing references that can only be used to read data. On the references that also allow to write data, the owner-as-dominator property is enforced.

Universe types discussed in this report both enforce the owner-as-modifier discipline. We will not discuss the owner-as-dominator property any further.

1.2 Universe Type System

The Universe Type System [11] is an ownership type system that enforces the owner-as-modifier discipline with little annotation overhead. To enforce the owner-as-modifier property, the Universe Type System allows you to modify objects that are either in the same context, or in the context directly owned by you. It introduces two *ownership modifiers* which precede references that allow to modify the object they are pointing to:

- `peer` denotes a reference pointing to an object in the same context as the current object (`this`)
- `rep` references must only point to objects directly owned by `this`

As stated, the owner-as-modifier property allows arbitrary references between objects, as long as they cannot be used to modify the referenced object. These are preceded by a third modifier:

- `any` annotates references that point to an object in an arbitrary context, but they must not be used to alter it (in earlier versions of the Universe Type System, `any` used to be called `readonly`)

In addition to reading from a reference, `peer` and `rep` references allow to write. Thus, `peer` and `rep` are subtypes of `any`. There is no subtype relationship between `peer` and `rep` since they denote orthogonal properties: One stays in the same universe whereas the other goes one step down in the hierarchy. The subtype relationship is illustrated in Figure 1.1.

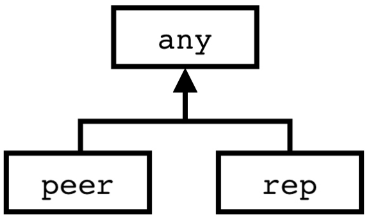

Figure 1.1: Subtype relationship between ownership modifiers: `any` is a supertype of `peer` and `rep`

Because methods can possibly alter objects, they must not be called on `any` references. This is a very severe restriction. Reference chains often go through many objects and information hiding encourages to provide getters instead of permitting direct access to the reference. Getters do not modify the object but just return a reference which would be safe. To solve that problem, it is legal to call *pure* methods on `any` references. A pure method is annotated with the `pure` keyword, and must not modify any objects. There exist different definitions of purity — we define that pure methods must not call non-pure[1] methods and must not alter fields of existing objects. It is allowed to use local variables, e.g. creating an iterator that is used to walk through a list and find the desired element.

1.2.1 Type Combinator

Ownership modifiers are always relative to `this`. Whenever a reference is accessed through another object, the type combination operator * needs to be applied to determine the correct ownership modifier relative to the origin of the access.

[1]Note that there is no keyword for non-pure methods. This is the default behavior.

*	**peer**	**rep**	**any**
peer	peer	any	any
rep	rep	any	any
any	any	any	any

Table 1.1: Type combinator for the Universe Type System

The type combinator is applied from left to right: For a field access `e.f`, i.e. field `f` is accessed through object `e`, we combine the ownership modifier of `e` with `f`'s ownership modifier. The resulting modifier denotes how `e.f` relates to the object this access occurred from.

1.2.2 An Example

The following example clarifies how the Universe Type System is used.

Listing 1.1: An example how the Universe Type System is used

```
 1  class UTSExample {
 2          class Node {
 3                  peer Node next;
 4                  any Object elem;
 5          }
 6
 7          rep Node head;
 8
 9          void add(any Object element) {
10                  rep Node newHead = new rep Node();
11                  newHead.elem = element;
12                  newHead.next = head;
13                  head = newHead;
14          }
15
16          pure any Object getHead() {
17                  return head.elem;
18          }
19  }
```

Listing 1.1 shows a Java class with universe annotations. Inner class `Node` has a `peer` reference to the next node of the list which means that the next and therefore all nodes reside in the same context. The reference of type `any Object` to the element can also refer to a `peer Object` or `rep Object` due to the subtype relationship.

On line 7, we define the reference pointing to the first node of the list. The `rep` modifier implies that the current instance of `UTSExample` (`this`) is the owner of the node, and therefore every node. The owner-as-modifier property ensures that no node can be modified, or deleted without going through `this`. The ownership structure is shown in Figure 1.2.

Method `add` adds an element to the list. Since it is not pure, it can only be called on `peer` or `rep` references. On line 10, a new instance of `Node` gets created. The `rep` modifier declares the receiver (`this`) to be the owner. On line 11 and 12, the type combinator needs to be applied since the access does not happen on `this`. On line 11, `newHead` is `rep` and `elem` is `peer`. The combination results in `rep` and because `head` is `rep` as well, the assignment is valid.

Method `getHead` is a declared `pure` and can therefore be called on `any` references. It performs no computations, calls no non-pure methods and does not make assignments.

The Universe Type System can cope with almost every feature the Java programming language offers, including interfaces, arrays, static members, exceptions and inner classes. It cannot cope with generic types though.

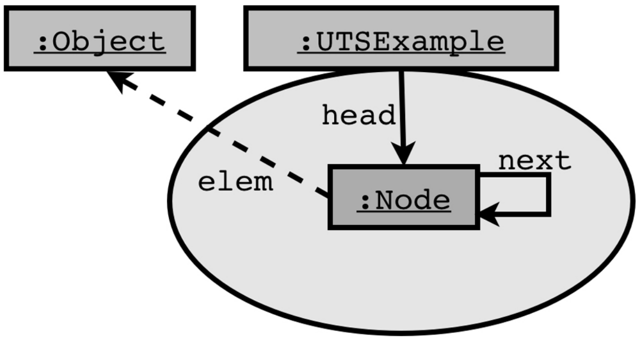

Figure 1.2: Object structure of the presented example: Instances of `UTSExample` own their `Node` objects due to the `rep` annotation. The nodes have a `peer` reference to the next node which therefore is in the same context, and they have an `any` reference to an `Object` in an arbitrary context. The dashed arrow depicts a read-only reference.

1.3 Generic Types

A data structure like a linked list or a map can contain arbitrary elements. These might have any type, be it number or string, but you hardly ever put both in the same instance of the data structure. Before generic types, there was no way to tell the compiler which type the elements are of. This meant every time an element was extracted from the data structure, a cast was necessary to get back a reference of a type it was possible to operate on. Casts are checked at runtime, i.e. they are statically unsafe and might lead to an error when executing the program.

It is already been a few years since generic types have been added to some of the most widely-used object-oriented programming languages, such as Java 5.0 or C# 2.0. Generic types look very similar to templates in C++, but the actual inspiration is *parametric polymorphism* as found in functional languages like ML and Haskell.

Generic types allow us to have data structures whose element type is more specific than the language's root of the type hierarchy, e.g. we can have a list of integers or a map of strings. By being specific about the type of the elements, the compiler is able to check that we only insert elements of the given type and thus can ensure that all elements in the data structure have this type. As opposed to non-generic classes where casts are needed to extract elements, extracting elements from generic classes is statically safe, i.e. we have stronger static guarantees.

Generic types are parameterized by so-called *type variables*. A type variable defines a type which can be used within the class declaring the type variable. Although we do not know the exact type when writing the class, we know that it will be some specific type. This allows assignments of references having the same type variable as their type. To instantiate a generic type, a *type argument* has to be given for each type variable of the class. For type checking, the type variables are substituted by their type argument for the given instance of the generic type.

Listing 1.2: An example for a generic class and how to use it in Java 5

```
class Container<E> {
        E element;

        E get() {
                return element;
        }

        void set(E element) {
                this.element = element;
        }
}
```

```
12
13 class GenericClassExample {
14         void main() {
15                 Container<String> containerOfString = new Container<String>();
16                 containerOfString.set(”SomeString”);
17                 String someString = containerOfString.get();
18
19                 Container<Integer> containerOfInteger = new Container<Integer>();
20                 containerOfInteger.set(new Integer(77));
21                 Integer int77 = containerOfInteger.get();
22         }
23 }
```

Listing 1.2 shows an example for a generic class and how it is used, in Java 5. `Container` is a generic class as you can see from its type variable `E`. `Container` is a simple data structure holding a reference to an object of some type `E` in field `element` (line 2). It has a getter (line 4) and a setter (line 8). Note that the type of `element`, the return type of method `get` and the type of parameter `element` of method `set` are all of type `E`. On line 10, you see an assignment of a reference of type `E` to another reference of type `E`. This is statically safe since all references of type `E` are instantiated with the same type for one instance.

Method `main` of `GenericClassExample` shows two possible usages of the generic class `Container`: once as a container for an element of type `String` (line 15) and once for an element of type `Integer` (line 19). Note the similarity of the syntax: The class is declared as `Container<E>`, i.e. its name is `Container` with one type variable `E`. On line 15, it is used as `Container<String>` where `Container` obviously refers to the class and `String` is the type argument for type variable `E`. This implicitly substitutes every occurrence of the type variable by its type argument for this instantiation, and allows to retrieve the stored reference without cast, i.e. statically safe (line 17 and 21).

Java 5 (and many other languages) not only allow classes to be generic — methods can have type variables as well. This allows methods to be generic and increases type safety, e.g. it is possible to relate the return type of a method to the type of one of its parameters. We call type variables that belong to a method *method type variables* as opposed to type variables of a class which we call *class type variables*.

Type variables can have *upper bounds*. An upper bound restricts the set of type arguments for this type variable to subtypes of the bound. Due to this restriction, we can use members of the upper bound on references of the type variable. Note that type variables without explicit upper bound have `Object` as their implicit upper bound.

Listing 1.3: An example for a generic method and a type variable with an upper bound

```
class Collections {
        static <T extends Comparable> void sort(List<T> list) {
                // ...
                Comparable c1 = ...
                Comparable c2 = ...
                if (c1.compareTo(c2) < 0) {
                        // ...
                }
                // ...
        }
}

class GenericMethodExample {
        List<Number> listOfNumbers = new List<Number>();

16          // fill the list ...
17
18          Collections.sort(listOfNumbers);
19          Collections.<Number>sort(listOfNumbers);
20  }
```

Listing 1.3 shows an excerpt of the static method `sort` defined in the class `Collections` of the Java API[2]. Its purpose is to sort a list of comparable elements.

Method type variable `T` has an upper bound, `Comparable`. This means that `T` can only be instantiated with subtypes of `Comparable`, i.e. types that implement the interface `Comparable`. This restriction is necessary to be able to actually sort the elements. Imagine, a variable of type `List<Object>` is passed to the method: The list cannot be sorted since there is no order defined between the different elements. Here, since we know that all possible type arguments for `T` implement `Comparable`, we can take two arbitrary elements, assign them to variables of type `Comparable` and call method `compareTo` to tell us which of the elements has to come first in the sorted list. Valid types for parameter `list` are, for example, `List<Number>` (as on line 18) and `List<String>`.

Although Java has an inference algorithm for method type variables, i.e. it automatically infers the types of all method type variables and checks if the call is valid, it is also possible to explicitly specify their type. The syntax is as shown on line 19.

1.4 MultiJava and JML

MultiJava [8] is an open source project that adds open classes and symmetric multiple dispatch to the Java programming language. Open classes allow programmers to add methods to existing classes without editing those classes, or having their source code. Multiple dispatch allows the code invoked by a method call to depend on the runtime type of all the arguments, instead of just the receiver. It comes with a compiler that translates MultiJava code in normal Java bytecode so it can be run on normal Java virtual machines. The compiler supports version 1.4 of the Java language — but also has full support for generic types[3] that were added in version 5 of Java.

The Java Modeling Language JML [18] is a specification language that can be used to specify the behavior of Java modules. It makes the Design by Contract [1] approach of Eiffel [3] available to Java. It is open source and builds on top of the MultiJava compiler, i.e. it extends it and adds additional checks to enforce the JML contracts such as preconditions, postconditions and invariants.

1.5 Examples

Throughout the report, we will apply every concept to a short example. These examples are usually independent from each other — except for Chapter 2 where we use one example to illustrate most concepts of Generic Universe Types. In many examples, we refer to classes `Data` and `ExtendedData`. These represent typical classes storing various properties with no business logic.

Listing 1.4: Shared code of the examples

```
class Data {
        Object property;

        Object getProperty() {
                return property;
        }
```

[2]It slightly differs from the actual method defined in the Java API. The reason is that it contains wildcards that will be introduced in Section 3.4.

[3]This includes raw types and wildcards.

```

	void setProperty(Object property) {
		this.property = property;
	}
}

class ExtendedData extends Data {
}
```

Listing 1.4 presents classes `Data` and `ExtendedData`. `Data` holds a reference to a property which is of type `Object` and has a getter and a setter. `ExtendedData` inherits from `Data` without adding any members — its purpose is to illustrate in the examples how subtype relationships are handled by the various concepts.

We omit access modifiers in the examples. We assume that you are familiar with the basic classes of the Java API like `Object`, `Number` and `List<T>`. You can find the documentation of the Java 5 API in [2].

1.6 Overview

In Chapter 2, we will first give an introduction to Generic Universe Types and then present the addition of features in Java and describe the implementation. Chapter 3 explains the concepts of raw types and wildcards, and describes their application to Generic Universe Types. An approach to infer ownership modifiers in unannotated Java programs is illustrated in Chapter 4. Chapter 5 concludes this report and gives ideas for future work.

Chapter 2

Generic Universe Types

In this chapter, we will explain Generic Universe Types. We give an informal description of the core concepts such as viewpoint adaptation in Section 2.2 and the subtyping rules in Section 2.3. We then add Java's concepts like arrays in Section 2.5, static members in Section 2.6 and exceptions in 2.7. Defaulting of ownership modifiers is explained in Section 2.9. The chapter is concluded by a description of the implementation in Section 2.10.

2.1 Basic Concepts

Based on the Universe Type System, a new type system was developed that includes generic types. It is called Generic Universe Types [9] and it is the first type system that combines the owner-as-modifier discipline with generic types. It gives even stronger static guarantees, yet has very little annotation overhead.

Simply spoken, Generic Universe Types does to a generic type system what the Universe Type System does to a non-generic type system. In a non-generic type system we can specify the type of a reference. Ownership gives us the power to state that some object a reference is pointing to is owned by us. With generic types, we are able to denote a collection of some specific type. Adding ownership again, we are able to tell that we own some collection and that we also own the objects in that collection.

In this chapter, we will go into detail and discuss how Generic Universe Types can be integrated into Java 5. We take the core as it is defined in [9] and then add the various concepts of Java. We use the same example throughout the chapter to explain the concepts behind Generic Universe Types: a map storing key-value pairs in a linked list. Apart from calling a getter with the key of the desired element, an iterator is available that allows to walk through all key-value pairs and access them sequentially. A data structure with an iterator is a very popular example in the world of ownership. Reason being that many ownership type systems cannot handle that case since there are actually two objects that want to access the data structure: itself and the iterator. We will show how Generic Universe Types handles this.

The keywords of Generic Universe Types are the same as with the Universe Type System. The ownership modifiers `peer`, `rep` and `any` are used to annotate references and state which context the referenced object resides in, or rather has to reside in. The keyword `pure` annotates methods that do not modify state.

2.1.1 Types

We have to deal with two kinds of types in Java: *primitive types* and *reference types*. Primitive types such as `int`, `boolean` or `byte` are value types, i.e. there exists no aliasing problems because they are copied by value. Therefore, they are not annotated. In the following, we only have to consider reference types.

For reference types, we have *non-variable types* and *type variables*. Non-variable types consist of an ownership modifier, a class name, and possibly type arguments. Type arguments might also have ownership modifiers. Type variables on the other hand have no explicit ownership modifier, as they are substituted by the type arguments when the generic class gets instantiated. You can see that reference types have an arbitrary number of ownership modifiers.

To instantiate types in Java, we need constructors. Constructors are not part of the core of Generic Universe Types, but we can easily add them. A constructor in Java unites the allocation and initialization of an object. So, it simply is the first method that is called on the newly allocated object. There is one restriction though: The types of the constructor's parameters must not contain `rep` modifiers. This is necessary because the `rep` modifier implies that `this` is the owner of the object and since the object is about to be created, it cannot already own another objects.

Listing 2.1: Generic class `Pair` designed to store a key-value pair

```
class Pair<PK, PV> {
        PK key;
        PV value;

        Pair(PK key, PV value) {
                this.key = key;
                this.value = value;
        }

        pure PK getKey() {
                return key;
        }

        pure PV getValue() {
                return value;
        }

        void setValue(PV value) {
                this.value = value;
        }
}
```

Listing 2.1 shows the first class of our map with iterator example. It presents a class `Pair` which stores a key-value pair. Key and value can be of arbitrary type as they are each typed by a type variable: `PK` is the type for the key `key`. `PV` represents the type for `value`[1]. The class offers getters for key and value. Since they do not modify state, they are annotated with the `pure` keyword which allows them to be called on `any` references. `Pair` also offers a setter for the value, i.e. it is possible to change the value assigned to a key. The initial key set by the constructor cannot be changed. This is reasonable as you can easily create new instances of `Pair`.

The following example for a type declaration of `Pair` gives you a first idea how generic classes are to be used in Generic Universe Types: `peer Pair<rep Integer, any Object>` denotes an instance that lives in the same context as the object it is declared in. This is due to the so-called *main modifier* which is `peer` in this example. It is the first ownership modifier of a type declaration and defines the context the object is in, relative to `this`. The same holds for the ownership modifier occurring in the type arguments: They are relative to the object they are declared in and not to the class or object they instantiate. In this case, the key (`rep Integer`) of the `Pair` has to be owned by `this` — the current `Pair` object — whereas the value (`any Object`) can be in an arbitrary context.

[1]We use `PK` and `PV` instead of `K` and `V` to eliminate ambiguity between type variable names. We will have more classes having two type variables for key and value.

2.1.2 Interfaces

Interfaces is another feature that has been added to Generic Universe Types in order to make them useful with Java. Their integration is straightforward: Ownership modifiers and `pure` annotations have to be taken into account when checking properties such as subtype relationships and implementations of methods.

Listing 2.2: Generic interface `Link` facilitating the use of a generic implementation of an iterator

```
interface Link<X> {
        pure X getNext();
}
```

Interface `Link` shown in Listing 2.2 is a simple interface. Its type variable `X` represents the return type of the only method `getNext`. The interface will be used by the generic implementation of an iterator that can iterate over any data structure whose data element class implements this interface.

Declaring a method `pure` forces the implementor of the method to stick to it, i.e. it must not modify existing objects or call non-pure methods. The same applies to methods overriding a pure method: It must again be pure. Since there is no keyword for a non-pure method, it is implicitly added if a method not being declared pure overrides a `pure` method. It is also possible to declare a constructor `pure` which then implies that no other object than the newly created one is modified.

Listing 2.3: Generic class `Entry` representing an entry in a map.

```
class Entry<EK, EV> extends Pair<EK, EV> implements Link<peer Entry<EK, EV>> {
        peer Entry<EK, EV> next;

        Entry(EK key, EV value, peer Entry<EK, EV> next) {
                super(key, value);
                this.next = next;
        }

        pure peer Entry<EK, EV> getNext() {
                return next;
        }

        void setNext(peer Entry<EK, EV> next) {
                this.next = next;
        }
}
```

Class `Entry` presented in Listing 2.3 extends `Pair` and implements `Link` so it can be used as a data element in a map, but at the same time use an iterator to walk through all entries of the map. It inherits the methods from `Pair` to read the key, to read and write the value, and adds a getter and a setter for `next`. The setter for `next` is not necessary to implement the `Link` interface, but it is useful to rearrange the elements in the map, e.g. to delete an entry.

One interesting fact to point out is the usage of `peer` instead of `any` as the main modifier for `next`. We could as well use `any` here because neither do we use the reference to modify the next element nor do we call non-pure methods on it. The reason is that we enforce all entries linking to each other to be in the same context, i.e. it is impossible to mistakenly link two entries that do not belong to the same map.

2.1.3 Bounded Type Variables

As in Java, specifying an upper bound for a type variable in Generic Universe Types forces the type argument to be a subtype of the bound. For references of a type variable type, the bound determines which operations are permitted. For example, if the main modifier of the upper bound is `any`, modifications are prohibited like for non-variable types whose main modifier is `any`. Note that the unbounded type variable has an implicit upper bound `any Object`, i.e. modifications are not allowed.

Listing 2.4: Generic implementation of an iterator using the `Link` interface

```
class Iterator<T extends any Link<T>> {
        T current;

        pure Iterator(T first ) {
                current = first ;
        }

        pure T getNext() {
                T tmp = current;
                current = current.getNext();
                return tmp;
        }

        pure boolean hasNext() {
                return current != null;
        }
}
```

Listing 2.4 shows a very basic iterator. `Iterator` can iterate over arbitrary subtypes of `any Link` as long as the type argument for `T` is the same as for `Link`'s type variable. This upper bound has an interesting property: It contains itself in the upper bound which is allowed by Java and Generic Universe Types. Since the main modifier of `T`'s upper bound is `any`, modifications to `current` are prohibited.

For type variables having `peer` as the main modifier of their upper bound, their references point to objects in the same context and therefore are safe to modify. On the other hand, `rep` must not occur in upper bounds of class type variables for the same reason `rep` modifiers are not allowed in constructors: Ownership modifiers of upper bounds are relative to the class in which they are declared. Since the class is just about to be instantiated, it cannot be the owner of objects.

Java allows to have multiple upper bounds, i.e. it is possible to specify up to one class and an arbitrary number of interfaces the type argument has to be a subtype of. Such types are called *intersection types*. The syntax in the presence of ownership modifiers is `T extends peer Data & Comparable<T>`, i.e. only the first bound has a main modifier. Although it is possible to give an ownership modifier for each upper bound, this is not desireable. It is sufficient to have the most specific ownership modifier at the beginning as they all define the same relationship of contexts. If multiple bounds have a main modifier and these modifiers are not subtypes of each other, the type variable is not instantiable. For that reason, we ensure that are main modifiers are subtypes of the first main modifier and apply the main modifier of the first bound to all bounds.

2.2 Viewpoint Adaptation

We have explained that ownership modifiers are always relative to the class or the object they occur in. For upper bounds of type variables, this means that they are relative to the generic class declaring the type variable. Type arguments are relative to where the generic type declaration occurs. This might be in a different class which means that the ownership modifiers have to be adapted in order to validate the instantiation of a type variable, similar to the type combinator in the Universe Type System. In Generic Universe Types, this extended form of combination is called *viewpoint adaptation*.

2.2.1 An Example

Viewpoint adaptation is needed when accessing a generic type through a chain of references. We will show how this works by adding the `Map` interface to our example, and then show possible usages.

Listing 2.5: Generic interface `Map` offering the two basic operations and an iterator

```
interface Map<MK, MV> {
        void put(MK key, MV value);
        pure MV get(MK key);
        pure peer Iterator<any Entry<MK, MV>> iterator();
}
```

A simple interface for a map is presented in Listing 2.5. `Map` has two type variables for the type of key and value. Method `put` adds a key-value pair to the map. Its counterpart `get` looks for the value that belongs to key `key` and returns it. The signature of `iterator` needs some explanation: The method returns an instance of `Iterator` that is in the same context as the `Map` itself. The elements it iterates over are the entries of the map, i.e. key and value are accessible. The exact type `any Entry<MK, MV>` implies that the iterator gives read-only access to map's entries. Thus, the non-pure methods `setValue` and `setNext` cannot be called which means that the map's internal structure cannot be modified although it is exposed. The downside is that safe modifications, such as changing a value, are prohibited as well. We will show a solution to that problem in Section 2.8.

Listing 2.6: An example where viewpoint adaptation is required

```
1  class ID {
2  }
3
4  class ViewpointAdaptation {
5          void main() {
6                  peer Map<rep ID, any Data> map = ...
7                  peer Iterator<any Entry<rep ID, any Data> iterator;
8                  iterator  = map.iterator();
9          }
10 }
```

Listing 2.6 lists two possible usages of `Map`. In the first case, on line 6, the map `map` is in the same context as the current instance of `ViewpointAdaptation`. The keys are objects of type `ID` that are owned by `this` and the values are `Data` objects that belong to an arbitrary context. Calling the `iterator` method requires viewpoint adaptation to determine the return type, i.e. we have to adapt the return type `peer Iterator<any Entry<MK, MV>>` from the viewpoint `peer Map<rep ID, any Data>` to the viewpoint `this`.

2.2.2 Adapting a Viewpoint

Adapting a type from a viewpoint to the viewpoint `this` comes down to combining the ownership modifier of the first type with the second type since, at first, we do not care about the first type's type arguments but only need to know the relationship of the contexts of the two types which is denoted by the main modifier of the first type. Keep in mind that the main modifier of the first type states the relationship between its context and the context of the object it refers to. The main modifier of the second type states the relationship between its context and the context it refers to. As you can see, viewpoint adaptation can be performed with two types, or with an ownership modifier and a type.

When combining an ownership modifier u with a type, i.e. potentially several ownership modifiers, we use the type combinator, as it was already defined and applied in the Universe Type

System, to combine ownership modifier u with each ownership modifier of the type. The type combinator for two ownership modifiers is shown in table 2.1.

	peer	**rep**	**any**
peer	peer	any	any
rep	rep	any	any
any	any	any	any

Table 2.1: Resulting ownership modifier when combining two ownership modifiers. It is equal to the type combinator from the Universe Type System shown in Table 1.1.

As a last step, we have to substitute the type variables by their type arguments. The type arguments belong to the first type and therefore do not need to be adapted since their viewpoint is already `this`.

Going back to the example from Section 2.2.1, this means combining type `peer Map<rep ID, any Data>` with type `peer Iterator<any Entry<MK, MV>>`. We first take the main modifier of the first type, `peer` and combine it with every modifier of the second type `peer Iterator<any Entry<MK, MV>>`. The result is the exact same type since `peer` combined with `peer` is again `peer` and `peer` with `any` is `any`. After substituting the type variables `MK` and `MV` by their type arguments `rep ID` and `any Data`, we get `peer Iterator<any Entry<rep ID, any Data>>` as type for the iterator `iterator`.

2.2.3 Situations Requiring Viewpoint Adaptation

We have presented how viewpoint adaptation is applied to the return type of a method when the method is not called on `this`. As you might expect, it works the same when a field is accessed on another reference: Instead of the return type of the method, the type of the field has to be adapted.

Viewpoint adaptation is also required to check the validity of the arguments of method and constructor calls, i.e. the viewpoint of the parameter type has to be adapted to the argument's viewpoint. For constructors, we combine the type of the object to be created with the parameter types. For method calls, we take the type of the receiver of the call and combine it with the parameter types. This is done for all parameters and checked with their corresponding argument.

Instantiating type variables needs viewpoint adaptation as well. We have to make sure that the given type argument is a subtype of the type variable's upper bound. Before we can perform the subtype check, the upper bound's viewpoint has to be adapted to the type argument's viewpoint. This is done by adapting the viewpoint of the upper bound to the viewpoint `this`.

We will show an application in the following example.

Listing 2.7: A simple implementation of the `Map` interface using a linked list

```
class LinkedMap<LK, LV> implements Map<LK, LV> {
        rep Entry<LK, LV> head;

        void put(LK key, LV value) {
                head = new rep Entry<LK, LV>(key, value, head);
        }

        pure LV get(LK key) {
                peer Iterator<any Entry<LK, LV>> iterator = iterator();
                any Pair<LK, LV> current;
                while (iterator.hasNext()) {
                        current = iterator.getNext();
                        if (current.getKey().equals(key)) {
                                return current.getValue();
                        }
                }

                return null;
        }

        pure peer Iterator<any Entry<LK, LV>> iterator() {
                return new peer Iterator<any Entry<LK, LV>>(head);
        }
}
```

`LinkedMap` presented in Listing 2.7 represents a simple implementation of the `Map` interface. Internally, it uses a linked list consisting of `Entry` objects to store key-value pairs. We have the pure method `get` retrieving a value from the map based on the passed key. It uses an iterator to walk through the internal list and return the value if its key matches the given key. Otherwise, `null` is returned. Note that, depending on the definition used for purity, method `iterator` must not be pure because it creates a new object.

In method `put` on line 5, a new instance of an `Entry` owned by `this` is created and the new entry is inserted at the beginning of the list[2]. To check the validity of this call, we have to combine the type of the object to be created with each parameter type. The most interesting case is the third parameter `next`. We combine `rep` with type `peer Entry<EK, EV>` and get `rep Entry<EK, EV>` as a result before substituting the type variables. The final result of the viewpoint adaptation is determined by substituting the type variables by their type arguments, which are type variables as well. We get `rep Entry<LK, LV>` as the viewpoint adapted type for the third parameter of the constructor. It remains to check that the argument of the call, `head`'s type `rep Entry<LK, LV>`, is a subtype of the viewpoint adapted parameter type. The types of the other two parameters of `Entry`'s constructor are type variables. The viewpoint of type variables does not need to be adapted as they are substituted by their type arguments, and type arguments are always relative to `this`.

Method `iterator` presents another case where viewpoint adaptation is required: When `Iterator`'s type variable `T` is instantiated, the type argument `any Pair<K, V>` has to comply with its upper bound `any Link<T>` as shown in Listing 2.4. Before we can check the subtype relationship, the upper bound's viewpoint has to be adapted to the type argument's viewpoint. We combine the type itself with the upper bound, i.e. `peer Iterator<any Pair<K, V>>` is combined with `any Link<T>` resulting in `any Link<any Pair<K, V>>`. In the next section, we will explain why this is a valid subtype.

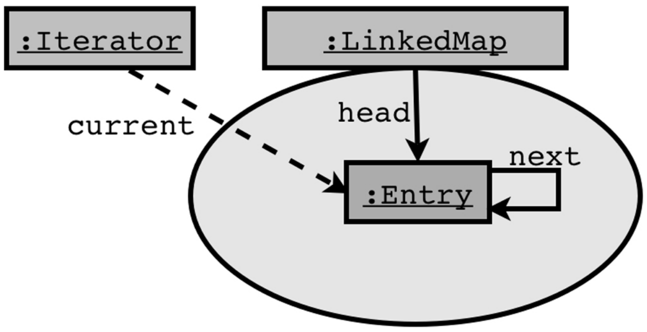

Figure 2.1: Ownership structure of `LinkedMap`, `Entry` and `Iterator`

[2]For simplicity, we assume that no two values with the same key are added to the map.

2.2.4 Preserving Type Safety

Viewpoint adaptation as we have presented it so far, is not type safe. We will illustrate this with the following example.

Listing 2.8: An extension to the example showing the problem with viewpoint adaptation as we have presented it so far

```
 1  class ExtendedIterator<T extends any Link<T>> extends Iterator<T> {
 2          void setCurrent(T current) {
 3                  this.current = current;
 4          }
 5  }
 6
 7  interface AlternativeMap<AK, AV> {
 8          // ...
 9          peer ExtendedIterator<rep Entry<AK, AV>> iterator();
10  }
11
12  class ID {
13  }
14
15  class AlternativeMapClient {
16          void main() {
17                  rep AlternativeMap<peer ID, peer Data> alternativeMap = ...
18                  rep ExtendedIterator<any Entry<peer ID, peer Data>> iterator;
19                  iterator = alternativeMap.iterator(); // invalid assignment
20                  iterator.setCurrent( /* ... */ );
21          }
22  }
```

Listing 2.8 presents a more powerful iterator named `ExtendedIterator` which extends the known `Iterator` and has a setter for the current entry of the iterator, i.e. it basically allows you to jump back and forth. Furthermore, we have an alternative interface for a map, called `AlternativeMap` which fits exactly the implementation of `LinkedMap`: It forces the implementor to be the owner of the entries by using the `rep` modifier, as opposed to `Map` which allows the entries to be in an arbitrary context due to the `any` modifier.

Calling `iterator` on `alternativeMap` means combining `rep AlternativeMap<peer ID, peer Data>` with `peer ExtendedIterator<rep Entry<AK, AV>>`. The result — based on what we have presented so far – is type `rep ExtendedIterator<any Entry<peer ID, peer Data>>`. On line 20, `iterator`'s `setCurrent` method now allows us to pass `any Entry<peer ID, peer Data>` since that is the type argument for type variable `T`.

This is obviously not type-safe and therefore wrong. It happened because `rep` combined with `rep` resulted in `any`. Although we actually know the owner, we cannot express it since we do not have an ownership modifier describing this relationship and we take `any`. The solution is, whenever a `rep` modifier occurs in one of the type arguments of the second type, to change the main modifier to `any`. This prohibits modifying the object and therefore ensures type safety. In the example above, the correct viewpoint adapted type is `any ExtendedIterator<any Entry<peer ID, peer Data>>`.

We present an alternative solution in Section 3.6.

2.3 Subtyping

In Figure 1.1, we have presented the relationship between the various ownership modifiers: `any` is at the top of the hierarchy and `peer` and `rep` are subtypes of it. `peer` and `rep` do not have any kind of subtype relationship. In this section, we are looking at subtyping in general.

The following basic rules apply to subtyping in Generic Universe Types:

- Subtyping is always stricter than in Java [14], i.e. if a type A is not a subtype of type B in Java, A is not a subtype of B in the presence of ownership modifiers either.
- Subtyping is still transitive, i.e. if B is a subtype of A and C is a subtype of B, C is also a subtype of A.
- Subtyping is still reflexive, i.e. A is a subtype of A.

In order for a type A to be a subtype of another type B, A's main modifier has to be a subtype of B's main modifier. For example, `rep Object` is a subtype of `any Object`, `peer ExtendedData` is a subtype of `peer Data` and `any Data` is a subtype of `any Object`. As a counter-example, `rep ExtendedData` is not a subtype of `peer Data`.

These rules are sufficient for non-generic types. When we add generics, the rules get slightly more complicated: Basically, the ownership modifiers of type arguments have to be equal in order for two types to be in a subtype relationship. For example, `peer List<peer Data>` is a subtype of `any List<peer Data>`, `rep List<rep ExtendedData>` is a subtype of `rep Collection<rep ExtendedData>`[3] and `peer List<rep Collection<peer Object>>` is a subtype of `any List<rep Collection<peer Object>>`. As counter-examples, `rep List<peer Object>` is not a subtype of `rep List<any Object>` and `peer List<rep Data>` is not a subtype of `peer Collection<any Data>`.

We can loosen the above equality constraint of all ownership modifiers by allowing a limited covariance. As opposed to Java where we have invariance, we can permit limited covariance with respect to ownership modifiers. The intolerable behavior with covariance is that a `List<ExtendedData>` can be seen as a `List<Data>` and `Data` objects can be inserted into the list of `ExtendedData` which is illegal. Let us now consider the following type: `any List<any Data>`. The main modifier is `any` which disallows modifications to the list. Since the behavior stated above cannot occur, we can allow covariance, e.g. `peer List<peer ExtendedData>` could be a subtype of `any List<any Data>`. This though contradicts one of the basic rules saying that a type not being a subtype of another in Java, is not a subtype in the presence of ownership modifiers either. Therefore, we limit covariance to ownership modifiers, i.e. `peer List<peer Data>` is a subtype of `any List<any Data>`. Note that the class may vary, e.g. `rep List<rep Data>` is a subtype of `any Collection<any Data>`.

The idea is to allow covariance for ownership modifiers if the supertype's main modifier is `any`. In the example, since `any List<any Data>` has `any` as its main modifier, the ownership modifier of `Data` in `peer List<peer Data>` can be `peer` and we still have a subtype relationship. Note that the main modifier of the subtype is irrelevant. As counter-examples, `any List<peer ExtendedData>` is not a subtype of `any List<any Data>` and `peer List<rep Object>` is not a subtype of `any Collection<peer Object>`. Note that this informal description using examples is vague. We refer to [9] for a formal definition.

2.4 Additional Rules

In the preceding sections, we have presented an almost complete overview of Generic Universe Types. In this section, we are going to add some rules necessary to ensure type safety.

The only ownership modifier that must occur in parameter types of pure methods is `any`. Why is that? We know that methods declared `pure` can be called on `any` references, i.e. from any context. Furthermore, we know that when we check the validity of the arguments, we first have to combine the receiver type with the declared parameter type to adapt the viewpoint before we can perform the subtype check. If the receiver type is `any`, the viewpoint adaptation will yield `any` which is not type-safe. Therefore, we require parameter types of pure methods to only have `any`

[3]`List<E>` is a subtype of `Collection<E>` in Java.

modifiers. Although this might look restrictive, it is not: Pure methods must not change state, and due to the subtyping rules (including limited covariance) we can still pass any type we could pass if other modifiers were allowed.

Similar to that, methods that are not called on `this` must not contain `rep` modifiers in their parameter types. Again, combining a type different from `this`[4] with a `rep` modifier yields `any` which then leads to not type-safe calls. The same applies to upper bounds of method type variables: As opposed to class type variables, method type variables may have `rep` modifiers in their upper bound. Because of that, we have to ensure that calls to such methods only happen on `this`.

The exact same problem also exists for field updates: Imagine you change a `rep` field through a `peer` or `rep` reference. The combination of the two is an `any` reference, thus allows you to assign a value of type `any` to it which is not type-safe. We use the same idea to prevent this from happening: `rep` references can only be changed on `this`.

Furthermore, we do not support method overloading based on ownership modifiers. In Java, it is possible to have two methods `void foo(Data d)` and `void foo(ExtendedData d)` with the same method name but different parameter types. The compiler then selects one of the methods based on the static type of its arguments[5]. One might now extend this by having two methods that are only different with respect to the ownership modifiers, e.g. two methods `void bar(rep Data o)` and `void bar(peer Data o)`. This is not supported. Methods that are that similar usually do the same, otherwise you might want to consider giving them a different name. If they do the same and you do not care whether you get an object in the same context or an object owned by you, you probably did something wrong in your ownership hierarchy which you have to fix.

2.5 Arrays

You are now familiar with the core concepts of Generic Universe Types. It is time to integrate the concepts of Java to make them work together. In this section, we are going to discuss arrays. Arrays were part of the Universe Type System, but are not included in the core of Generic Universe Types.

On the one hand, an array is an object itself. On the other hand, it is a collection of values or references, depending on whether the component type is a primitive type or a reference type. For arrays of primitive types, the meaning of the ownership modifier is clear: It denotes the context the array object is in. An example is `peer int[] intArray`, where `intArray` is an array in the same context as `this` and whose component type is `int`. Since `int` is a primitive type, values are copied and there is no need for an ownership modifier for the component type.

Arrays of reference types are different: Not only do we need to know which context the array object resides in, but we also need to know where the objects referenced by the array are. Thus, arrays of reference types have two ownership modifiers. The first denotes the context of the array itself — the second states the context of the referenced objects. One questions remains: Is the second ownership modifier relative to `this` or to the array? In the Universe Type System, it was decided the second ownership modifier be relative to the array, i.e. `rep peer Object[] objArray` declares an array that is owned by `this` and whose referenced objects are owned by `this` as well since they are in the same context as the array itself. Figure 2.2 shows the ownership structure. This interpretation of the second ownership modifier renders `rep` as ownerhsip modifier for the component type useless: An array has no operations and thus cannot be the owner of objects.

This interpretation was reasonable for the Universe Type System, but it would be inconsistent with Generic Universe Types. When thinking about ownership modifiers in Generic Universe Types, we know that their viewpoint is always the place where the are declared. `rep Collection<peer Object> objColl` denotes a collection `objColl` owned by `this`. The elements are in the same context as `this`, i.e. `this` and the collection's elements have the same owner. This hierarchy is shown in Fgure 2.3.

[4]Combining `this` with an ownership modifier u yields u.
[5]MultiJava even selects the method based on the runtime type.

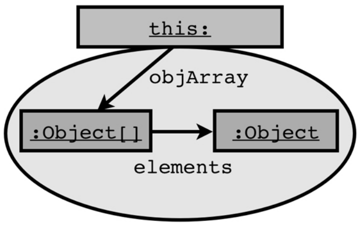

Figure 2.2: Ownership structure of `rep peer Object[] objArray` as it was interpreted in the Universe Type System

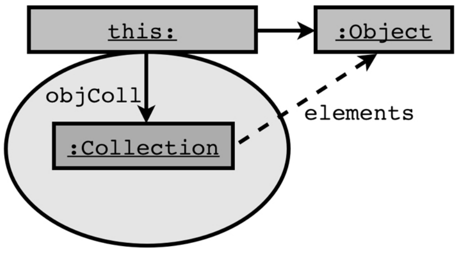

Figure 2.3: Ownership structure of `rep Collection<peer Object> objColl`

Comparing the two types `rep peer Object[]` and `rep Collection<peer Object>`, we see that they look similar and should have the same ownership structure for consistency. But their ownership diagrams are different due to the interpretation of the component modifier of the array type. This is misleading and might cause confusion, especially when changing the interpretation from an array to a generic `Collection`, or vice versa. Thus, for Generic Universe Types, we changed the interpretation of the second ownership modifier to be interpreted relative to `this` as well which solves the problem. Figure 2.4 shows the new ownership structure of the new interpretation and one can see that it now looks equal to the one of the collection. Correctness of the new semantics has been proven by Martin Klebermaß in his Master's thesis [17].

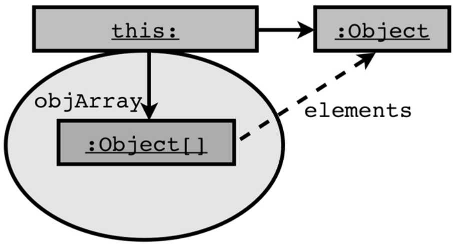

Figure 2.4: Ownership structure of `rep peer Object[] objArray` as it is interpreted in Generic Universe Types

Another case to consider are arrays whose component type is a type variable such as `T[] genericArray`. Although its component type is a reference type, such an array only has one ownership modifier: `peer T[] genericArray` is in the same context as `this`. The second ownership modifier is part of the type variable `T` which is relative to where the type argument for the type variable is defined.

2.6 Static Members

The keyword `static` can be used with various Java constructs: fields, methods, inner classes and inner interfaces. From a universe point of view, static inner classes and static inner interfaces are not very interesting as they are simply another way of organizing your code. Instead of putting them in their own file, they are embedded in another class or interface.

Static fields are more interesting. A static field is a reference that is available per class instead of per object as it is the case for non-static fields. Although it is possible to access static fields through an instance, it is recommended to access them through their class name. Static fields are used for example to implement the singleton pattern [13] in Java.

Since static fields belong to no instance, we do not know which context they are in and therefore which objects can modify the objects they reference. The simplest approach is to not allow modifications through static fields at all, i.e. to treat them as `any` references. Other possible approaches for the Universe Type System are discussed in [15].

Disallowing modifications through static fields sounds like a severe restriction, but most static fields in Java are declared `final` which prohibits assigning another object to such a reference. From that you can see that changing static references is discouraged. Note the difference between `any` and `final`: `any` forbids modifications through the object, e.g. its fields, whereas `final` does not allow changing the object assigned to the reference. Good programming style tries to avoid global variables, and static fields basically are global variables. Relying on them can be bad with respect to multi-threading, and especially in Java where we can have different class loaders or the code running on several virtual machines concurrently, static is not always static [20].

Static methods are not bound to an instance, either. They can be called from every context without having an instance of the class they belong to. The idea is that static methods are executed in the same context as the caller. For that reason, static methods cannot be called on `any` references, i.e. we need to have a current context[6]. In the signature of static methods, `any` and `peer` may occur where `peer` refers to the context of the caller. `rep` on the other hand is forbidden since we do not have a `this` object in a static context and cannot own any objects. Note that this rule also applies to types inside a static method's body and to static initializers.

Listing 2.9: Class `Maps` offering a method that drops duplicates in a map

```
class Maps {
        static <K, V> peer Map<K, V> dropDuplicates(peer Map<K, V> map) {
                peer Map<K, V> result = new LinkedMap<K, V>();
                peer Iterator<any Entry<K, V>> iterator = map.iterator();
                while (iterator.hasNext()) {
                        any Pair<K, V> current = iterator.next();
                        if (result.get(current.getKey()) == null) {
                                result.put(current.getKey(), current.getValue());
                        }
                }
                return result;
        }
}

class StaticCall {
        void main() {
                peer Map<rep Integer, any Object> peerMap = ...
                rep Map<rep Integer, any Object> repMap = ...
                // insert key−value pairs ...
                peerMap = peer Maps.<rep Integer, any Object>dropDuplicates(peerMap);
                repMap = rep Maps.<rep Integer, any Object>dropDuplicates(repMap);
```

[6]The current context is important to initialize objects. For dynamic casts, it is necessary that new objects are created in a specific context.

```
        }
}
```

Listing 2.9 gives an example for a static method and how to call it. We define a class `Maps` that — similar to the `Collections` class in the Java API — provides static methods that either modify the existing collection or create a new collection with a certain property. Here, we give the implementation for a method `dropDuplicates` whose purpose it is to create a new map having the same key-value pairs but all keys are unique. This is done by creating a new map, iterating over all entries of the passed map and add each key-value pair that does not yet exist in the newly created map. To check if a key already exists in the new map, we try to retrieve the value that belongs to it. If it is `null`, we know that the key does not yet exist[7].

Class `StaticCall` shows the two ways a static method can be called: in the current context, i.e. in the context `this` belongs to, or in the context owned by `this`. To call the method in the current context, `peer` has to be added before the name of the class – to call it in the context you own, `rep` has to be used. This triggers viewpoint adaptation, i.e. `peer` or `rep` is combined with the parameter types and the return type of the static method.

As you can see from the example, we explicitly specify the type arguments for the method type variables although we mentioned in Section 1.3 that an algorithm exists that can automatically infer type arguments for method type variables. This algorithm is designed for Java and is inherently complex as it has to deal with the various features Java offers in terms of generics such as multiple upper bounds and wildcards[8]. Although we suspect that it can be adopted — probably with slight changes — to Generic Universe Types, we have not done this as the original algorithm is missing in the compiler we did our implementation. For more details on the implementation, we refer to Section 2.10.

2.7 Exceptions

In this section, we are going to discuss how to ensure type safety in Generic Universe Types in the presence of exceptions. Exceptions are meant for cases when something unlikely happens, e.g. if a connection to a server cannot be established or a buffer overflows. Bad programmers also tend to use them as a form of control flow. Apart from the type of the exception, they mostly carry information telling why it has been thrown.

There are two ways to deal with them: They can be caught using a `catch` statement and then handled, e.g. attempt to reconnect or flush the buffer. If it is unknown how to recover from the exception, it can also be rethrown. For so-called *unchecked exceptions* this happens automatically if no `catch` clause is present. For *checked exceptions*, a `throws` statement has to be added to the method signature.

From this description you can see that, when an exception is thrown, we have no idea where or how it will be handled, because we usually do not know who will call the method that throws the exception. And, an exception is hardly ever modified. It contains information about the error case but it does not offer methods to recover from it.

The simplest solution to deal with exceptions is sufficient: All exceptions are read-only by default, i.e. their ownership modifier is `any`. In Java, exceptions must not be generic due to the lack of generic support at runtime[9]. So, this solution — which is the same as for the Universe Type System — is fine for Generic Universe Types as well. With exceptions being read-only, we do not care in which context they end up. Since there usually is no need to modify exceptions, this is not too restrictive either.

Alternative solutions for the Universe Type System and other ownership type systems are discussed in [10].

[7]We assume the value of a key-value pair to be not `null`.
[8]Wildcards are covered in Section 3.4.
[9]We will talk about runtime support of generic types in Chapter 3.

2.8 Multiple Modifying Objects

Classes `Pair` and `Entry` offer setters for the value and the next entry which have not been of use yet. Since we are enforcing the owner-as-modifier discipline, these methods can only be called by the owner or peers. In the implementation of `Map` that we are using, the map itself is the owner of the entries. This is sensible if we want to prove invariants about the map's entries. In the following, we are showing how the map itself can make use of the iterator to modify its entries.

Listing 2.10: `ExtendedLinkedMap` adds methods to replace a value and to delete a key

```
class ExtendedLinkedMap<K, V> extends LinkedMap<K, V> {
        peer Iterator<rep Entry<K, V>> internalIterator() {
                return new peer Iterator<rep Entry<K, V>>(head);
        }

        void replace(K key, V value) {
                peer Iterator<rep Entry<K, V>> iterator = internalIterator();
                while (iterator.hasNext()) {
                        peer Pair<K, V> current = iterator.getNext();
                        if (current.getKey().equals(key)) {
                                current.setValue(value);
                        }
                }
        }

        void delete(K key) {
                peer Iterator<rep Entry<K, V>> iterator = internalIterator();
                rep Entry<K, V> previous, current;
                while (iterator.hasNext()) {
                        current = iterator.getNext();
                        if (current.getKey().equals(key)) {
                                if (previous != null) {
                                        previous.setNext(current.getNext());
                                } else {
                                        head = current.getNext();
                                }
                        } else {
                                previous = current;
                        }
                }
        }
}
```

`ExtendedLinkedMap` shown in Listing 2.10 subclasses `LinkedMap` and adds functionality to replace a value and to delete a key. These methods are used to demonstrate how the same iterator that cannot alter the map from the outside, can be used internally to modify it. Note that these methods are not available through the `Map` interface.

Method `internalIterator` is used internally to retrieve an iterator that allows to modify the `Entry` objects. The difference to the iterator providing access the map from the outside is `Entry`'s ownership modifier: For internal use it is `rep` whereas for external use it is `any`. The necessity of this distinction is explained in Listing 2.8 and its explanation. In short, this is needed since viewpoint adaptation would render the internal iterator useless. Thanks to the `rep` reference, we can call non-pure methods to modify the entries.

Method `replace` uses the internal iterator to walk through the list of entries and replace the value of each entry with the given `key` by the new `value`. Method `setValue` is used to update the value.

In method delete, each entry having a specific key is deleted. Again, the internal iterator is used. Entry's method setNext is called on the previous entry to have its next reference point to the next entry after the one to be deleted. Java's garbage collector will automatically remove the unused entry.

You might now be wondering if it was possible to have an iterator that could be used to modify the entries from the outside, e.g. to assign a different value to a key. The answer to that question is somewhat complicated. The owner-as-modifier discipline requires to go through o's owner for every modification of object o. Applying this property to the Entry objects means that each modification has be initiated by the corresponding Map, i.e. the iterator can definitely not directly modify the entries. But, since the iterator and the map are in the same context, the iterator could call non-pure methods on the map that altered its entries. If we wanted to change the value of an entry, this would require the iterator to pass the entry and the new value to the map which then updates it.

It is not that simple though: The iterator only has an any reference to the Entry which it can pass on. The map needs a read-write reference, i.e. in this case a rep reference. A cast is the only way to solve this but, this is not statically checkable and therefore not desirable. Alternatively, the map could look up the entry in the list and then modify it. In terms of runtime, this comes down to the same as calling the replace method.

Another approach is to have peer references to the entries, i.e. instead of owning its Entry objects, they reside in the same context as the map, and therefore the iterator. This allows both, the map and the iterator, to modify them. This solution results in a less deep ownership hierarchy and therefore is not suited to prove certain invariants and properties. As you can see, there is always a trade-off between strong static guarantees (deep ownership structure) and flexibility (flat ownership structure).

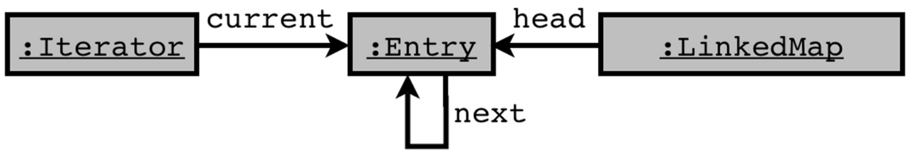

Figure 2.5: Alternative ownership structure for LinkedMap, Entry and Iterator

2.9 Defaulting

Although Generic Universe Types already have very little annotation overhead, it can be further reduced by using appropriate defaults. These defaults are applied whenever an ownership modifier is missing, or it was intentionally left out because the default is fine. Apart from reducing the annotation overhead, defaulting is designed to make normal Java code, i.e. code without ownership annotations, valid with respect to the rules of Generic Universe Types.

This is achieved by putting all objects in the root context. Based on that, peer is the obvious choice as the default ownership modifier. Basically, all missing ownership modifiers are implicitly set to peer, e.g. List<Data> becomes peer List<peer Data>. New instances are also implicitly annotated with peer, e.g. new Object() becomes new peer Object() which ensures that new objects are created in the same context and therefore, peer references are statically correct.

As opposed to type arguments, the ownership modifiers for upper bounds of type variables default to any. This choice is reasonable since most objects being accessed through a reference whose type is a type variable, are just read or passed around, e.g. elements in collections. Therefore, peer would be too restrictive. Furthermore, for type variables without upper bound, this yields any Object as an upper bound which is more general than peer Object.

References to boxing types of Java's primitive types, e.g. Integer for int or Boolean for boolean, as well as String default to any. Since they are immutable, this is a reasonable choice.

For exceptions, defaulting is different: As mentioned, these are read-only by default since they cross universe boundaries. Therefore, the reference in the `catch` clause defaults to `any`, e.g. `catch (Exception e)` becomes `catch (any Exception e)`. As well as static fields. Since they do not belong to a specific context, they default to `any`, e.g. `static Object o` becomes `static any Object o`.

The default modifier for method signatures of pure methods is `any`, e.g. `pure Data getDescription(Object o)` becomes `pure any Data getDescription(any Object o)`. Furthermore, methods overriding a pure method automatically become pure as well. A very special case are arrays of reference types with only one ownership modifier. In such a situation, this modifier becomes the component (second) modifier and the main (first) modifier defaults to `peer`.

Defaulting allows to annotate Java programs in multiple steps. Beginning with a valid program that has all objects in the same context, a deeper ownership hierarchy can be created by adding more and more contexts. These give you a better understanding and better control of the program. In Chapter 4 we discuss how a deep universe hierarchy can be inferred.

2.10 Implementation

We have implemented all static checks of Generic Universe Types as presented in this chapter[10] in the MultiJava [8] compiler `mjc` which makes it available to the JML tools [18]. Note that changes to JML were minimal. In combination with the dynamic checks that are currently being implemented by Mathias Ottiger [22] as part of his Master's Thesis, you have full tool support for Generic Universe Types. This section will only deal with static checks.

We know that Generic Universe Types is a superset of the Universe Type System. The only difference is the interpretation of the second ownership modifier for arrays of reference types (see Section 2.5). Therefore, we built our implementation on top of the existing implementation for the Universe Type System which was done in MultiJava as well. We do not expect issues with the change in the array semantics as these rare cases can be changed quickly.

2.10.1 General Design Decisions

On each passed source code file – called *compilation unit* – `mjc` performs several passes, i.e. it executes the task on each compilation unit. Examples for passes are parsing, checking correctness of signatures or typechecking methods.

For our implementation, we intended to add a new pass at the end that performs all checks that are necessary to ensure the type rules of Generic Universe Types – so-called *universe checks*. The reason was that the implementation for the Universe Type System performed their checks in several passes, i.e. the checks were distributed across the code base and therefore hard to understand. After taking a closer look at the situation, it turned out that this approach had some considerable disadvantages:

- The passes change the abstract syntax tree and the semantic table that prevent us from performing viewpoint adaptation. For example, the typechecking pass substitutes type variables by their type arguments which makes viewpoint adaptation impossible: We do not know if a type has to be adapted, or if it is a substituted type variable that does not need to be adapted.

- All the passes share is the abstract syntax tree and the semantic table. This means that a lot of information that is computed by other passes had to be recomputed as it is not stored. This adds additional complexity to the implementation and also slows down the compiler.

Due to these reasons, we decided to continue performing universe checks along the way in other passes. Yet to make our part of the implementation easier to read, understand and maintain, we factored out as much of the code as possible in classes only dealing with the semantics of Generic

[10] The only exception is arrays whose component type is a type variable, e.g. `T[]`.

Universe Types. All classes of Generic Universe Types are prefixed by `CUniverse`. We also maintained a uniform naming style for methods performing universe checks that are placed in common code.

2.10.2 Changes to the Syntax

Thanks to the existing implementation for the Universe Type System, changes to the syntax were minimal: What we refer to as main modifiers was already allowed. All we had to do was allowing ownership modifiers in declarations of type variables (namely the upper bounds) and type arguments. The definitions of the syntax dealing with these two occurrences are `jTypeVariableDeclaration` and `jTypeParameter`.

Storing the ownership modifiers was straightforward: `CClassType`, the base class of all reference types, contained a field named `universe` that stored the ownership modifier. For upper bounds of type variables and type arguments, which are subtypes of `CClassType` as well, this field was not used. With Generic Universe Types it now is.

Another important change to the syntax was the introduction of the `any` keyword. We added it to the rule `mjUniverseReadonlySpec`. Throughout the code, you will not find references to the `any` keyword. In class names and error messages we still refer to it as `readonly`.

For the lexer and parser, MultiJava uses ANTLR [4] which allows to have Java code in the parser definition. Therefore, we have some tasks for Generic Universe Types that are coded in the grammar `Mjc.g`:

- Defaulting of upper bounds to `any` happens in rule `jTypeVariableDeclaration`. If the type variable has no explicit upper bound, we use `any Object`. If the type variable has multiple upper bounds, we ensure that all main modifiers are assignable to the main modifier of the first bound and apply the first main modifier to all other bounds.
- The array semantics for the Universe Type System prohibited the second array modifier from being `rep`. With the new semantics, this is legal. We removed the checks in `jClassTypeSpec` and `jNewExpression`.

2.10.3 Viewpoint Adaptation

Before we implemented viewpoint adaptation, we had to add an additional ownerhsip modifier: The `this` modifier is unknown to the Universe Type System. In Generic Universe Types, it is used to represent the self-references `this` and `super`. We added class `CUniverseThis` as a subtype of the base class of all ownership modifiers, `CUniverse`. All subclasses of `CUniverse` are singletons.

The type combinator of the Universe Type System was implemented in `CUniverse`. It offered a static method named `combine` which took two ownership modifiers and returned the combination of them.

In Generic Universe Types, the type combinator is known as viewpoint adaptation, and is a lot more complicated. We decided to give it its own class: `CUniverseViewpointAdaptationService`. As the name suggests, it is a service class which means that it is not instantiable and only has static methods. That way, its functionality is available to all classes.

Viewpoint adaptation is implemented in `CUniverseViewpointAdaptationService` exactly as it is formally specified in [9] in Section 3.2. We define the overloaded method `combine` and a few helper functions:

- `CUniverse combine(CUniverse first, CUniverse second)` combines the ownership modifier `first` with `second` and returns the resulting modifier. It is the implementation of $\triangleright$ :: $\mathtt{OM} \times \mathtt{OM} \to \mathtt{OM}$.
- `CClassType combine(CUniverse modifier, CClassType type)` combines the ownership modifier `modifier` with `type` and returns the resulting type. It implements $\triangleright$ :: $\mathtt{OM} \times {}^{s}\mathtt{Type} \to {}^{s}\mathtt{Type}$.

- `CUniverse combineM(CUniverse modifier, CClassType type)` is a helper function to adapt a type with respect to a ownership modifier. It returns the correct main modifier. It implements $\triangleright_m$ :: OM $\times$ ^{s}NType $\rightarrow$ OM.
- `boolean containsRepModifier(CClassType type)` looks for the `rep` modifier in the given type. It returns true if there is at least one. This method is overloaded: It also takes an array of `CClassType`s and checks all of them for the `rep` modifier.
- `CClassType combine(CClassType first, CClassType second)` combines the type `first` with the type `second` and returns the resulting type. It is the implementation of $\triangleright$ :: ^{s}NType $\times$ ^{s}Type $\rightarrow$ ^{s}Type.

There is one difference to the formal specification: Method `combine` adapting a type w.r.t. a type does not substitute the type parameters of a generic type by their type arguments. Instead it just ignores them because they are substituted along the way when it is necessary for further checks. Yet, we have to substitute part of the type variables before performing viewpoint adaptation: Since we did not define lookup functions as they are outlined in Section 3.4 of [9], we need to substitute type variables of inherited members by their instantiations in the current class. This is done by method `substituteInheritedTypeVariables` which modifies the type to look as if it were defined in the current class, i.e. the type returned by method `substituteInheritedTypeVariables` equals the type returned by the lookup functions.

The `combine` methods do not modify the passed types but clone them using Java's `clone` method. This is necessary as the same type might be used in multiple statements of the abstract syntax tree. These might have to be adapted to different viewpoints which we do not know and therefore do not want to change.

2.10.4 Subtyping

Subtype relationships of types are checked in method `descendsFrom` of class `CClass` although the method actually looks at types, not classes. The reason is that MultiJava had already existed before generic types were added to Java. Due to that, generics are not completely well integrated, but rather built on top.

To add the subtyping rules of Generic Universe Types, we had to integrate them into the existing methods that check the subtype relationship in Java. If we had decided otherwise and had built our own subtyping algorithm, this would have required to traverse the class hierarchy twice for each subtype check.

Furthermore, it is not enough to apply subtyping rules of Generic Universe Types when universe checks are turned on, and use Java's subtyping rules otherwise. For example, when looking up a method based on its receiver type, name and the types of the arguments, we use Java's subtyping rules for various reasons: First, we would have to perform viewpoint adaptation on the method's parameter types to check for a subtype relationship. This is time-consuming when done often. Second, we want to be able to give different error messages depending on whether the error is on the Java side, or it is the ownership modifiers that are wrong. This is not possible if the lookup returns unsuccessfully and we have no idea why no method was found.

To accommodate these requirements, we had to add a switch to the methods that tell which rules to apply. Most methods that deal with subtype checks now have an additional parameter called `enableUniv` of type `boolean` serving as the switch. In order to not break existing code, the method with the old signature still exists. It calls the new method and sets `enableUniv` according to the compiler's settings.

In some cases, the methods already had multiple parameters that served as switches to turn certain checks on or off. We then created a new method with the same name and added the suffix `NoUniverse` to indicate that this is the version that does not consider ownership modifiers.

The following classes and their methods implement the main subtype checks for Generic Universe Types. Note that there exist several versions for most listed methods. We also had to change or add implementations of these methods in subclasses of the classes given below.

- `CClass`: method `descendsFrom`
- `CClassType`: methods `isAlwaysAssignableTo` and `isValidTypeArgumentFor`
- `CTypeVariable`: methods `equals` and `isAlwaysAssignableTo`

We know that our implementation is far from best-practice. We did it anyway because a proper solution would have required us to spend a multiple of time on it. And we feared that we could thereby break existing functionality and we were not willing to take that risk.

2.10.5 Universe Checks

Having viewpoint adaptation and subtyping checks available, we can go on to implement the type rules of Generic Universe Types. Note that [9] distinguishes between type rules and well-formedness. We refer to both as rules.

Each class of the abstract syntax tree has a method `typecheck` being called in the typecheck pass. The vast majority of universe checks are performed in this pass. Thus, we have created a method named `typecheckUniverse` in each class which is called by the `typecheck` method if universe checks are turned on. Our method enforces rules for Generic Universe Types without confusing them with Java's checks. In the following, we list all classes that have such a method and explain briefly which checks are performed:

- `JClassFieldExpression` represents an access to a field. Viewpoint adaptation is applied.
- `JMethodCallExpression` stands for a method call. Various checks are required to ensure the method is called on a valid receiver and method arguments are valid. Furthermore, the return type needs viewpoint adaptation.
- `JAssignmentExpression` represents an assignment. We make sure that the right hand side is a subtype of the left hand side. Furthermore, the target must not be an `any` reference. Note that this check is also performed for primitive types.
- `JCastExpression` is an explicit cast. As suggested in [9], we check that the expression is actually castable to the target type. Due to an extremely complicated structure of the `typecheck` method, we did not factor it out into a separate method.
- `JNewObjectExpression` creates an instance of a class. We have to check that the constructor call is valid, and that the main modifier is not `any`.
- `JMethodDeclaration` declares a method. We ensure that all parameter types of pure methods only contain `any`. Furthermore, if the method overrides another method, each parameter's type has to be subtypes of the corresponding parameter's type of the overridden method. In the signature of static methods, no `rep` modifier must occur.
- `JConstructorDeclaration` declares a constructor. Checks are the same as in `JMethodDeclaration`.
- `JVariableDefinition` declares a variable. In a static context, no `rep` modifier must occur.

Although we really to place all checks in separate methods, we had to integrate some in existing methods. The following classes and methods contain further checks:

- `CTypeVariable#checkTypeUniverse` ensures that upper bounds of class type variables do not contain `rep` modifiers.
- `CClassNameType#checkTypeArguments` forces type arguments to respect their upper bounds.

2.10.6 Testing

Test cases covering the new functionality have been integrated into MultiJava's testing framework. Being placed in `org.multijava.mjc.testcase.universes.gut`, they are automatically run when `make runtests` is called. Test cases consist of at least one Java class with universe annotations and mostly contain several test cases for one aspect, e.g. inheritance, arrays or viewpoint adaptation. Each test case additionally has one file that contains the messages the compiler is supposed to produce.

Apart from the new tests, there exist lots of tests for the Universe Type System. These had to be updated due to the changed array semantics and updated error messages (to be found in `CUniverseMessages.msg`).

Despite all these test cases and having run additional tests, we cannot give any guarantee for the correctness of the implementation. The universe checks have been tested thoroughly, but support for generic types was added recently to the MultiJava and thus occasionally contains bugs. We have added the ones we found but did not fix to the MultiJava bugs database [11].

2.10.7 Usage

MultiJava [8] — which includes the MultiJava compiler `mjc` — is open source and can be downloaded from the project's website. It has command line flags to turn on universe (`-e`) and generics (`-G`) support. When installed properly, it is used as follows: `java org.multijava.mjc.Main -e -G <input files>`, i.e. you run the `main` function of class `Main` in the package `org.multijava.mjc` and pass it the two flags and the input files.

As soon as you start using classes from the Java API in your universe code, you will have to use JML. Since these classes have no ownership annotations, defaulting is applied. If you wanted to call method `equals` through an `any` reference, you will get an error saying that you must not call a non-pure method through an `any` reference. To circumvent this, JML comes with and allows you you to write specifications for classfiles, i.e. classes that are only available in byte code as `.class` files. In the subdirectory `specs` of JML's project folder, the package hierarchy is reproduced and searched whenever an external class is loaded. If a specification file is available, these universe annotations are taken instead of the defaults, e.g. the specification for `Object`'s `equals` method states that it is pure. After a successful installation, it can be run by `java org.jmlspecs.checker.Main -e -G <input files>`.

[11] `http://sourceforge.net/tracker/?atid=511776&group_id=65658`

Chapter 3

Raw Types and Wildcards

In this chapter, we will cover raw types and wildcards and their application to Generic Universe Types. Section 3.1 explains erasure. In Section 3.2, we describe raw types in Java before we apply them to Generic Universe Types in 3.3. Wildcards in Java are explained in Section 3.4. In Section 3.5, we show how wildcards can be used with Generic Universe Types. The chapter is concluded by an application of the wildcard concept to ownership modifiers in Section 3.6.

3.1 Erasure

Before explaining what raw types are, we will take a look at the design of generics in Java 5. Although generics look very similar in every object-oriented programming language, handling of generic types varies considerably. The C++ template mechanism creates a new class for every concrete parameterization. While offering great flexibility and high efficiency, it can lead to code bloat. From that perspective, the C++ implementation can be seen as a kind of high-level macro.

For C#, Microsoft decided to provide full runtime support for generic type information with version 2.0 of .NET, i.e. they changed the virtual machine to make parameterized types first-class objects. Sun took a different route for Java: The Java virtual machine (JVM) does not know about generics, i.e. they are a language-only construct. They are implemented in the compiler and the generated classfiles contain generic signatures only in the form of metadata which allows the compiler to compile new classes against them. The runtime has no knowledge of the generic type system which meant that JVM implementations only needed minimal updates to handle the new class format.

The advantage is that old and new code work together seamlessly[1]. This is especially important in regard to the Java API. If old Java code were not compatible with new generic code, the JVM would have to come with two versions of the API: a non-generic and a generic version. And both these version would have to be maintained. Otherwise, all existing Java code — which is a lot — would not run on computers with a newer version of the Java virtual machine since a compatible version of the API classes was missing.

That is achieved by dropping all generic type information when source code gets compiled into byte code. This mechanism is called *erasure* — denoted by $|\,|$ — and maps generic types to non-generic types as follows [14]:

- The erasure of a parameterized type $G < T_1, ..., T_n >$ is $|G|$.
- The erasure of a nested type $T.C$ is $|T|.C$.
- The erasure of an array type $T[]$ is $|T|[]$.
- The erasure of a type variable is the erasure of its leftmost bound.

[1] A new version of the byte code format was introduced which forces you to update your JVM to run Java 5 byte code.

- The erasure of every other type is the type itself.

We show how erasure works by applying it to the following example:

Listing 3.1: An example for a simple generic class `Container`

```java
class Container<T extends Data> {
	T element;

	T get() {
		return element;
	}

	void set(T element) {
		this.element = element;
	}
}
```

Listing 3.2: Erased version of class `Container` presented in Listing 3.1

```java
class Container {
	Data element;

	Data get() {
		return element;
	}

	void set(Data element) {
		this.element = element;
	}
}
```

Listing 3.1 presents a simple generic class with a type variable `T` upper bounded by class `Data`. The erased version of the same class is shown in Listing 3.2. You can see that in the erased version, no generic information is left, i.e. the type variable disappeared and was replaced by its upper bound. These two classes would look the same on the byte code level, apart from the generic meta information that is stored in the classfile of the generic version.

Think about how you would have written the class `Container` if generics had not been available? The answer matches the erased version: Instead of using a type variable, we take the most general type we want to allow.

Erasure not only affects the generic class, but also its clients. The type arguments of reference types are dropped. Instead, casts are inserted where needed, i.e. the compiler inserts exactly the same casts that were inserted by hand before generic types.

Listing 3.3: An example how `Container` can be used

```java
class SampleClient {
	void main() {
		Container<ExtendedData> c = new Container<ExtendedData>();
		c.set(new ExtendedData());
		ExtendedData ed = c.get();
		Data d = c.get();
	}
}
```

Listing 3.4: Erased version of class `Container` presented in Listing 3.3

```
1 class SampleClient {
2         void main() {
3                 Container c = new Container();
4                 c.set(new ExtendedData());
5                 ExtendedData ed = (ExtendedData) c.get();
6                 Data d = (Data) c.get();
7         }
8 }
```

In Listing 3.3 a possible usage of class `Container` is shown. After creating a new instance of a container holding `ExtendedData`, a new instance is set and then the element is retrieved twice. First it is assigned to an instance of `ExtendedData`, second to a reference of `Data`.

Since all classes are erased at compile-time, Listing 3.4 presents the erased version of `SampleClient`. You can see that all type arguments are dropped. On lines 6 and 7, casts get inserted by the compiler. Interestingly, it is exactly the casts you would have inserted if generics had not existed. Note that the cast on line 6 is not necessary.

You can see from this example that in Java, generics are really language-only and the compiler basically produces the code you have written before generics were introduced. Despite that, it still offers full type-safety of generic types at compile-time.

At runtime, not all checks and casts can be performed safely. To explain this issue, we define types that are fully supported at runtime — so called *reifiable types* — and types that are merely supported at compile-time. Basically, types that do not lose information by applying erasure, are reifiable. The following types do lose information and therefore are not reifiable:

- Type variables
- Parameterized types unless all actual type arguments to the type are unbounded wildcards[2]
- Array types whose component type is not reifiable

Casts and `instanceof` checks can only be partially checked for not reifiable types, i.e. only the erased type is checked. In such cases an *unchecked warning* is issued at compile-time. For example, `o instanceof List<String>` can only check that `o` is an instance of `List`, but it has no information on the type argument.

Other drawbacks are that reflection is only possible for reifiable types, i.e. no reflection is possible for generic types. Furthermore, only reifiable types can be instantiated, e.g. `new T()` is illegal for type variable `T`.

3.2 Raw Types

A *raw type* is the name of a generic type declaration used without any accompanying actual type arguments. For example, `Container container` declares a raw type since `Container` has a type variable as you can see in Listing 3.1 but no type argument is provided for it. Therefore, all declarations of types from the Java Collections API became raw types when they were run on version 5 of the Java platform.

With erasure in mind, it becomes obvious how raw types work. For raw types, type checking is performed against the erased version of the class, i.e. generic type information is ignored. This is exactly as it was handled before generics were introduced, because the erased version of the class equals its non-generic counterpart.

The use of raw types is strongly discouraged because there is no reason for not specifying type arguments. They were only introduced to maintain backward compatibility to existing code and to allow to move from non-generic code to generic code in multiple steps. Imagine you used

[2] Wildcards are discussed in Section 3.4.

a framework to write your program, e.g. the Java Collections API. Thanks to raw types, you could switch to the generic version of the framework as soon as it was released without having to change your code. Then, you could step by step introduce type arguments and therefore make your program statically safer.

3.3 Integrating Raw Types into Generic Universe Types

To discuss the application of raw types to Generic Universe Types, we will apply defaulting as described in Section 2.9 to the example used to explain erasure. It will reveal why we need to change defaulting for raw types in order to have them work properly with Generic Universe Types for unannotated programs.

Listing 3.5: Erased version including ownership modifiers of class `Container` presented in Listing 3.1

```
 1  class Container {
 2          any Data element;
 3
 4          any Data get() {
 5                  return element;
 6          }
 7
 8          void set(any Data element) {
 9                  this.element = element;
10          }
11  }
```

If class `Container` presented in Listing 3.1 is used with Generic Universe Types, defaulting is applied as there are no ownership annotations given. All that gets annotated is the upper bound of `T` which becomes `any Data` instead of just `Data`. When it gets compiled, erasure is applied and the resulting class is presented in Listing 3.5. All occurrences of type variable `T` have been replaced by its upper bound `any Data` as erasure is defined, i.e. method `get`'s return type is `any Data`.

To show a possible usage of `Container` as a raw type, we use an example similar to Listing 3.4 — the erased version of `SampleClient` from the erasure example.

Listing 3.6: An example using `Container` as a raw type

```
1  class RawTypeClient {
2          void main() {
3                  Container c = new Container();
4                  c.set(new ExtendedData());
5                  ExtendedData ed = (peer ExtendedData) c.get();
6                  Data d = c.get();
7          }
8  }
```

The difference of Listing 3.6 to Listing 3.4 is on line 6: Since the version presented in Listing 3.4 originated from a generic class by applying erasure, a cast was inserted even though it is not necessary for type safety. The code from example was written by a programmer. He knows that `Container` returns a reference of type `Data`. Therefore, he will not use a cast to assign it to a reference of type `Data`. Let us now use this class with Generic Universe Types.

Listing 3.7: Example from Listing 3.6 with default ownership modifiers made explicit

```
1  class RawTypeClient {
2          void main() {
3                  peer Container c = new peer Container();
4           c.set(new peer ExtendedData());
5           peer ExtendedData ed = (peer ExtendedData) c.get();
6           peer Data d = c.get(); // illegal
7       }
8   }
```

Using class `RawTypeClient` with Generic Universe Types means applying defaulting. Listing 3.7 presents the example from Listing 3.6 after defaulting has been applied. You can see that defaulting puts all objects in one context and inserts `peer` where an ownership modifier is required. On line 3, a `peer Container` instance is created and on line 4, it is set to point to a new `peer` instance of `ExtendedData`. On line 5, the cast's main modifier defaults to `peer` which makes the assignment valid because `c.get()` returns an object of type `any Data`. The assignment is statically and dynamically correct. On line 6, not considering ownership, no cast is needed as `Data` is the type `c.get()` returns. When taking ownership modifiers into account, this is not true anymore: We are trying to assign a reference whose static type is `any Data` to a reference of type `peer Data`.

This violation results from different defaulting for upper bounds of type variables. In the combination with erasure, what would normally default to `peer` becomes `any`. The obvious solution that comes to mind is to automatically insert casts where needed. Finding these situations where such a cast is required might be difficult. Therefore, we will in the following present an alternative.

The alternative is to automatically convert all raw types to generic types. By adding type arguments, the compiler no longer looks at the erased version of the class but compiles it against the generic version which is correct with appropriate type arguments. With generic types, casts are automatically inserted when erasure is applied. It then comes down to finding raw types which is easier than finding situations where casts are required. As type argument, we take the corresponding type variable's upper bound and replace the ownership modifiers by `peer`. In the example, `c` becomes of type `peer Container<peer Data>`. We can be sure that this instantiation is valid as the only two legal ownership modifiers for upper bounds are `peer` and `any`.

3.4 Wildcards

In this section, we will discuss wildcards — another concept introduced in Java 5 that has to do with generic types. In the subsequent section, we are going to show how this concept can be applied to Generic Universe Types.

We have mentioned in Section 2.3 that subtyping with respect to type arguments is not covariant in Java. Since it is not contravariant either, it is invariant. For example, although `ExtendedData` is a subtype of `Data`, `List<ExtendedData>` is not a subtype of `List<Data>` (no covariance) and `List<Data>` is not a subtype of `List<ExtendedData>` either (no contravariance). This is unlike arrays which are covariant: `ExtendedData[]` is a subtype of `Data[]`.

Listing 3.8: Example why covariance is not type-safe

```
1   class Covariance {
2       void main() {
3           LinkedList<Data> list = new LinkedList<ExtendedData>(); // illegal
4           list.add(new Data());
5
6           Data[] array = new ExtendedData[1];
7           array[0] = new Data(); // ArrayStoreException
8       }
9   }
```

In Listing 3.8 we give an example showing why generic types are invariant in Java. If the assignment on line 3 were valid, we could insert an instance of `Data` into a list of `ExtendedData`. In this example, we also show how Java deals with covariance of arrays: The assignment on line

6 is valid, but as soon as we try to store an object of `Data` in the array of `ExtendedData` an `ArrayStoreException` is thrown. Since it is an unchecked exception, it will most likely terminate the program, unless we have some exception handling mechanism or expect this to happen and catch it explicitly.

With this in mind, imagine you wanted to write a static method that takes an arbitrary list and reverses it in place. What would the method signature look like? A first solution one might come up with is `static void reverse(List<Object> list)`. This works fine for lists of `Object` but cannot be called with lists of other element types since they are not subtypes.

A more sophisticated approach uses a type variable to specify the list's element type: `static <T> void reverse(List<T> list)`. Thanks to the inference algorithm for method type variables, `T` can be any type which allows us to reverse lists of arbitrary element types. Although this solves the problem, it could still be simplified.

The only reason we introduced a type variable is because we do not know of another way to specify that we do not care about the type argument. This is what wildcards are for: They are simplest way to express that the actual type argument is irrelevant to us. A wildcard is expressed by the question mark `?` and states that we do not care about the type argument at all. It can be any type. The method signature for `reverse` then looks as follows: `static void reverse(List<?> list)`.

What we have presented so far are so-called *unbounded wildcards*. As with type variables, wildcards can also have bounds. The syntax for upper bounded wildcards is straightforward, e.g. `List<? extends Number>`. The upper bound gives us partial knowledge of the type, i.e. we know it is a subtype of the bound and therefore, can assign it to a reference of type `Number` in this case.

Additionally, wildcards can also be used in conjunction with a *lower bound*. As the name suggests, a lower bound expresses that the concrete invocation has to be a supertype, e.g. for `List<? super Number>` the type argument has to be a supertype of `Number`. This form of restriction is only supported by wildcards and not by type variables.

Although it is more rarely used than upper bounds, it provides us with even greater flexibility as you can see from an example taken from the Java Collection API: `static <M> void fill(List<? super M> list, M object)`. This method replaces every element of the list by the given `object`. These semantics are solely captured by lower bounded wildcards as `object` must be supertype of the list's element type.

3.4.1 Subtyping in the Presence of Wildcards

We have already explained why generic types in Java are invariant. Here, we will show the subtyping rules with wildcards. To that end, we introduce a relationship between type arguments called *type containment*. As defined in [14], a type argument A is said to contain another type argument B, written $B <= A$, if the set of types denoted by B is a subset of the set of types denoted by A. The type containment rules for concrete type arguments and wildcards are as follows:

1. ? extends $T <=$? extends S, if T is a subtype of S
2. ? super $S <=$? super T, if T is a subtype of S
3. $T <= T$
4. $T <=$? extends T
5. $T <=$? super T

Following this definition, an instantiation of a generic type is a subtype of another instantiation of the same generic type if and only if the former's type arguments are pairwise contained in the latter's type arguments.

From first two rules, we can see that upper bounded wildcards are covariant with respect to their bound whereas lower bounded wildcards are contravariant with respect to the bound. The thrid rule expresses what we have already known: In the absense of wildcards, the type arguments have to be equal in order for two generic types to be in a subtype relationship. The last two rules bridge between wildcards and concrete type arguments.

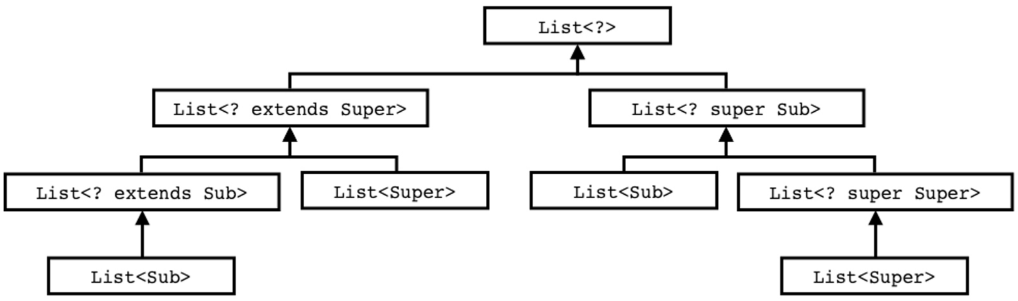

Figure 3.1: Subtyping among generic types with wildcards where `Sub` is a subtype of `Super`

In Figure 3.1 we find an example where the type containment operator is applied. We see that the unbounded wildcard is at the top of the hierarchy as it is the least specific, i.e. it contains no information at all about the type. This is also the reason why a wildcard type is reifiable as we pointed out in Section 3.1. The left branch of the tree shows covariant suptyping with respect to the upper bound and the right branch is contravariant with respect to the lower bound. Although the subtype relationship is depicted as a tree, it actually is not: Concrete instantiations of a generic type are a subtype of different instantiations with wildcards.

3.4.2 Capture Conversion

Wildcards by themselves are not real types. The set of types a wildcard covers also depends on the corresponding type variable and its upper bounds. Likewise, a bounded wildcard represents all types which fulfill the constraints imposed by both, the type variable's upper bounds and the wildcard's upper or lower bound. This dependence on a concrete type variable is the reason why wildcards can only be used as type arguments and their use as types of references would be meaningless.

Listing 3.9: Example showing that wildcards also depend on the corresponding type variable

```
1  class Wildcard<X extends Data> {
2          X x;
3
4          public void method1(Wildcard<?> wc) {
5                  Data d = wc.x;
6                  // ...
7          }
8
9          public void method2(Wildcard<? super ExtendedData> wc) {
10                 wc.x = new ExtendedData();
11                 Data d = wc.x;
12                 // ...
13         }
14 }
```

In Listing 3.9, we have a generic class `Wildcard` with one type variable `X` which is upper bounded by some class `Data`. Remember that a type variable cannot have a lower bound. Our class has a field `x` whose type is given by type variable `X`. On line 4, we see a method taking a reference to a `Wildcard` object. The method does not care about the actual type argument for `X`, thus an unbounded wildcard is used. Yet, it is legal to assign `wc`'s field `x` to a reference of type `Data`.

On line 9, we have another method taking a `Wildcard` object. Again, a wildcard is used for the type argument. In this case though it is lower bounded by `ExtendedData` which we define to be a subclass of `Data`. Thanks to the lower bound, we can now assign a new object of type `ExtendedData` to `wc`'s field `x`. And, we are still allowed to read `x`'s value and to assign it to a `Data` reference.

Why is all this valid? Every time a reference, whose type contains top-level wildcards, is read, *capture conversion* is applied. Capture conversion substitutes all top-level wildcards by a special kind of type variable that can have multiple upper bounds and at most one lower bound, i.e. it takes all upper bounds of the corresponding type variable and the bound of the wildcard, if any, and creates a new type variable with them. All other types are left untouched.

This has a few implications: For each substituted wildcard, a *new* type variable is created. Therefore, each wildcard is a different type, i.e. if a method signature contains two wildcards, these are completely independent. Furthermore, only wildcards of types being read — and not wildcards of types being written — are substituted by type variables. This allows assignments to types containing wildcards as long as the subtype relationship is respected. That is also true for non-top-level wildcards of types that are read: Since they remain so-called *unbound*, they can be assigned.

With this in mind, let us go back to the example: We have not explained yet why the assignment on line 5 is valid. In order to read `x`, `wc` has to be read first. Before this happens, capture conversion gets applied to `wc`'s type `Wildcard<?>`. The wildcard `?` is substituted by a fresh type variable with the upper bounds from the corresponding type parameter `X extends Data` and the wildcard's bound (none in this case). This results in `Wildcard<T_01>` where `T_01` is a newly created type variable with an arbitrary name and upper bounded by `Data`. Since `x`'s type is upper bounded by `Data` it is safe to assign it to `d`.

Let us also consider the example on lines 10 and 11. In this case the wildcard is substituted by a type variable that is lower bounded by `ExtendedData` — which makes line 10 a valid assignment valid — and upper bounded by `Data` — which makes line 11 valid. Notice here that capture conversion creates a type that cannot be expressed by the Java syntax, i.e. it is not possible to define a type variable that has both, upper bounds and a lower bound.

3.5 Integrating Wildcards into Generic Universe Types

Now that you have an in-depth understanding of wildcards in Java, it is time to apply them to Generic Universe Types. So far, we have had two kinds of types in Generic Universe Types: non-variable types and type variables. With the addition of wildcards, we introduce a third kind of types: wildcard types. In this section, we will show how they are integrated and show the rules.

3.5.1 Viewpoint Adaptation

If viewpoint adaptation has to be performed with wildcard types, we have to distinguish between two cases. Wildcard types can either occur on the left side of the operator, or on the right side. A possible third case where the types on both sides of the operator contain wildcard types, is covered by combining the approaches for the first two cases.

For the case where the type on the right side contains wildcard types in its type arguments, the viewpoint of the wildcard's bound gets adapted by being combined with the left type. As part of the viewpoint adaptation, we recursively check for `rep` modifiers in type arguments of lower bounds and change the appropriate modifier to `any` if there occur `rep` modifiers. We do not check upper bounds of occurring type variables, and therefore we do not check upper bounds of wildcards either as they do not allow modifications. We do check lower bounds though because such wildcards were designed for insertions which can violate the ownership structure.

Now let us look at the situation when the left side of the operator is a wildcard type. By combining two types, we adapt the second type from the viewpoint described by the first type to the viewpoint `this`. What we need to know is how the ownership of the first type relates to `this` which is described by the first type's main modifier. For non-variable types, this is obvious. For type variables, we take the main modifier of the first upper bound.

The left side of the operator is always the type whose member is accessed, i.e. it is read. Before a wildcard type is read, capture conversion is applied and a new type variable is created for each wildcard type and substitutes it. For normal type variables, we know to take the main modifier of the first bound in order to adapt the second type's viewpoint. Type variables created by capture conversion can have multiple upper bounds and up to one lower bound. Their main modifiers may be inconsistent which is why we take the most specific main modifier of all upper bounds, i.e. either `peer` or `rep` if one of them occurs, otherwise `any`. Note that `peer` and `rep` must not occur as main modifiers in bounds of the same type variable.

Note that wildcard capture requires viewpoint adaptation. The bound of the wildcard already has the correct viewpoint `this`. The viewpoint of corresponding type variable's bounds is the class they are declared in. To adapt their viewpoint to `this`, we need to combine the current type with each bound.

The following example shows applications of these rules:

Listing 3.10: An example to demonstrate viewpoint adaptation on wildcard types

```
 1  class Container<T extends any Data> {
 2          T element;
 3
 4          pure T get() {
 5                  return element;
 6          }
 7
 8          void set(T element) {
 9                  this.element = element;
10          }
11  }
12
13  class ViewpointAdaptationWildcards {
14          peer Container<? super rep ExtendedData> container;
15          peer ViewpointAdaptationWildcards vaw;
16
17          void main() {
18                  any Container<? super any ExtendedData> c = vaw.container;
19                  any Object o = container.get().getData();
20          }
21  }
```

In Listing 3.10 we present several examples for viewpoint adaptation involving wildcard types. We make use of class `Container` that is already well-known from previous examples.

The first access requiring viewpoint adaptation is found on line 18. The type on the right side contains a wildcard: `peer ViewpointAdaptationWildcards` is combined with `peer Container<? super rep ExtendedData>`. For the main modifier of the resulting type, we have to look at all type arguments to see if no `rep` modifier occurs in non-variable types and lower bounds of wildcards. The only lower bounded wildcard contains a `rep` modifier and therefore we have to set the main modifier to `any`. Next, we combine the first type's main modifier `peer` with the lower bound resulting in `any ExtendedData`. The viewpoint adaptation yields `any Container<? super any ExtendedData>`. You can see that if we had not changed the main modifier to `any`, it would have been allowed to call method `set` on `c` and insert `any ExtendedData`.

On line 19, we need to apply viewpoint adaptation twice. First where method `get` is called on field `container`. Since `container` is read, capture conversion is applied to its type `peer Container<? extends rep ExtendedData>`. The only top-level wildcard is replaced by a new type variable — we call it `T_01` — that is lower bounded by `rep ExtendedData` (no viewpoint adaptation required) and upper bounded by `any Data` (result of combination of `container`'s type `peer Container<? super rep ExtendedData>` with the `T`'s upper bound `any Data`). The viewpoint adaptation of `get`'s return type `T` from viewpoint `peer Container<T_01>` to `this` yields the newly created type variable `T_01`.

On the same line, `getData` is called requiring again viewpoint adaptation. Thereby the result from the first combination, `T_01`, is combined with `peer Object`, the return type of `getData`. To find the most specific main modifier of the first type, we look at all upper bounds of `T_01` which are only `any Data`. We take `any` and combine it with `peer Object` resulting in `any Object`.

3.5.2 Subtyping

Subtyping in Generic Universe Types with wildcard types comes down to applying the type containment operator as defined in Section 3.4.1 with the subtyping rules of Generic Universe Types.

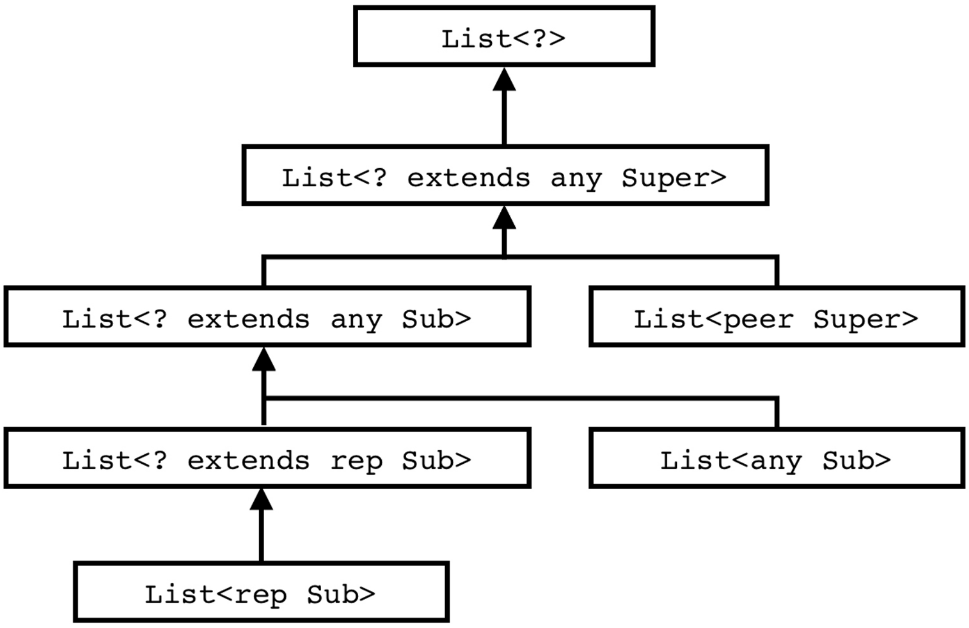

Figure 3.2: Subtyping in Generic Universe Types with upper bounded wildcards where `Sub` is a subtype of `Super`

Figure 3.2 gives an example for some types with upper bounded wildcards in Generic Universe Types. As you can see, upper bounded wildcards are still covariant with respect to their upper bound which includes the ownership modifier for Generic Universe Types. As an example, since `rep Sub` is a subtype of `any Sub` in Generic Universe Types, `? extends rep Sub` contains every type that `? extends any Sub` can adopt and therefore, we have a subtype relationship between `List<? extends rep Sub>` and `List<? extends any Sub>`.

Figure 3.3 is the equivalent to the previous figure with lower bounded wildcards. As in Figure 3.2, we have omitted the main modifier as it might be distracting. Arbitrary ownership modifiers can be used as main modifiers as long as there is the same subtype relationship between them. Furthermore, it is to note that these figures do not show the entire hierarchy but just give some examples.

Wildcards and limited covariance have in common that they both allow a subtype relationship between generic types with different type arguments. We will use a simple example to illustrate the

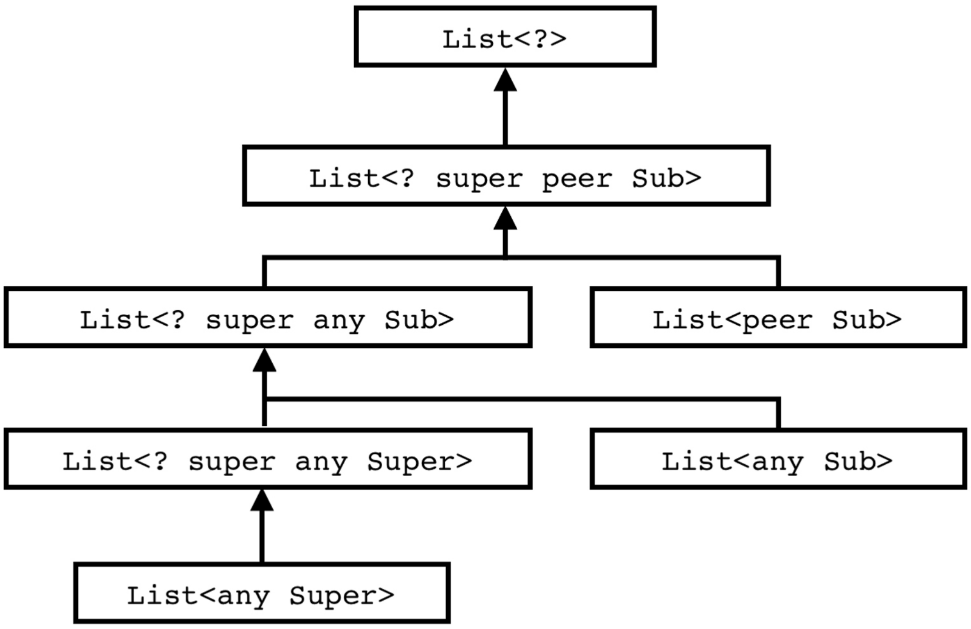

Figure 3.3: Subtyping in Generic Universe Types with lower bounded wildcards where `Sub` is a subtype of `Super`

difference between limited covariance and wildcards. The two types `peer List<rep Object>` and `peer List<any Object>` are in no subtype relationship because the type arguments are different and limited covariance only applies if the supertype's main modifier is `any`. On the other hand, if we change the type arguments to wildcards, there is a subtype relationship: `peer List<? extends rep Object>` is a subtype of `peer List<? extends any Object>` since `rep Object` is a subtype of `any Object`. Therefore we have a type containment and the type parameters are compatible, although they are different and the main modifiers are still `peer`. You can see that wildcards allow greater flexibility as they are also applicable for types with main modifiers different from `any`.

Due to the application of the subtyping rules of Generic Universe Types for type containment, we can also combine wildcards and limited covariance. If the wildcard's bound is a generic type with `any` as the main modifier, we can make use of limited covariance. Imagine you had the type `peer List<? extends any List<any Object>>`, i.e. a list of lists. This can be assigned an object of type `peer List<peer List<rep Object>>` due to limited covariance for the type argument.

Lower bounded wildcards are a new construct to Generic Universe Types. When looking into them, we thought we had to introduce a new ownership modifier. Imagine you had a lower bounded wildcard and you only cared about the class but not the actual ownership modifier, e.g. a list whose elements are a supertype of some type `Data`. In Java, the correct type is `List<? super Data>`, but what is it in Generic Universe Types? If you used `List<? super any Data>`, it would not be possible to assign a reference of type `List<peer Data` or `List<rep Object>` as lower bounded wildcards are contravariant with respect to their bound.

The solution would be to introduce a new ownership modifier that is a subtype of all existing ownership modifiers. But we did not do that as we think that this is a very rare case that does not justify a new ownership modifier. If you think about it: A lower bound is always relative to some other type because they are used to insert elements into a collection. Therefore, it is likely that the lower bound is a type variable and for these, this problem does not exist as we will see in the examples in Section 3.5.3. Furthermore, if you ever encounter a case where the lower bound is not a type variable and you are in a static method, you can simply use `peer` as the ownership modifier since `rep` is not allowed in static contexts and `peer` therefore is the lower bound in terms of ownership modifiers.

3.5.3 Examples

To illustrate the similarity between wildcards in Java and wildcards in Generic Universe Types, we will in the following show how to annotate a few methods taken from the Java Collections API:

- `static void reverse(List<?> list)`: The only annotation is the main modifier of `list` which has to be `peer` since `rep` is not allowed in a static context and `any` would forbid modifications to the list. The unbounded wildcard implies that the type argument can be anything and therefore also the ownership modifier, as an unbounded wildcard is basically upper bounded by `any Object`. Since no modifications are made to the list's elements, this is fine with Generic Universe Types. The resulting signature is `static void reverse(peer List<?> list)`.

- `static <T> void copy(List<? super T> dest, List<? extends T> src)`: Here, two annotations are required for the main modifiers of both lists. For the same reason as above, we choose them to be `peer`. Thanks to the way we treat static methods, the lists can reside in any context. All that is required is that the two lists are in the same context, i.e. they have the same owner. As the elements are not modified, this method works fine. Notice that the elements cannot be modified because `T` has no upper bound that would allow it. So, the signature is `static <T> void copy(peer List<? super T> dest, peer List<? extends T> src)`.

- `static <T> int binarySearch(List<? extends T> list, T key, Comparator<? super T> c)`: This method has a complex signature but a very simple purpose: It looks for the element `key` in the sorted list `list` by using the comparator `c`[3]. As you can imagine from the method's purpose, there is no need to modify any object. Thus, we choose all modifiers to be `any`, i.e. the main modifier for `list` and `c`. Does this work? Yes. All that is done is retrieving elements from the list and comparing them to the key using the comparator which are both `pure` operations and can be called on `any` references. The method's signature in Generic Universe Types is `static <T> int binarySearch(any List<? extends T> list, T key, any Comparator<? super T> c)`[4].

As you can see from the presented examples, annotating wildcard types is no big deal. Most of it is already taken care of by the defaulting mechanism. We have looked through parts of the Java API and have not found a method, that could not be annotated with ownership modifiers.

3.6 Universe Wildcards

Now that we have covered how the concept of wildcards can be used with Generic Universe Types, we take it one step further and have a look at how wildcards can be applied to ownership modifiers. The idea of what we call *universe wildcards* is to be able to express that we do not care about the ownership modifier of a type.

Basically, that is already covered by the `any` modifier because it is a supertype of all the ownership modifiers, but this does not work for type arguments. Let us assume you wanted a list you can modify and whose elements are of type `Data`. In Java, this is easily expressible as `List<Data>`, but how about Generic Universe Types? Since the list has to be modifiable, we can take `peer` or `rep` as the main modifier. The modifier of the type argument can be `any`, `peer` or `rep`

[3] A comparator is a comparison function, which imposes a total ordering on some collection of objects. Other than having elements of a type which extends `Comparable<T>`, you can have multiple comparators for the same list and therefore multiple ways of putting them in order.

[4] We did not declare `binarySearch` to be `pure` because static methods have to be called in a specific context, i.e. on a `peer` or `rep` reference.

but none of them allows any flexibility because it has to be an exact match as limited covariance only applies if the main modifier is `any`.

One approach to allow arbitrary ownership modifiers in the type argument is to use a wildcard. The type `peer List<? extends any Data>` allows the ownership modifier of the type argument to be any of the modifiers. Yet, this is only a partial solution as we restricted the main modifier to `peer` which disallows lists we own and the element's type is not limited to `Data` anymore but to any subtype of it. So, this can rather be seen as the equivalent for `List<? extends Data>`.

A proper solution offer universe wildcards. Simply spoken, this means applying the concept of wildcards to ownership modifiers instead of entire types. Unlike types, the set of ownership modifiers is limited. Thus, we choose a simple syntax and basically introduce two new ownership modifiers:

- *readable* `rd` can represent an arbitrary ownership modifier, be it `any`, `peer` or `rep`
- *writable* `wr` can represent one of the two ownership modifiers `peer` and `rep`

Having these two additional modifiers, the solution in the example above becomes obvious: `wr List<rd Data>`. By using `wr` as the main modifier, we support both, `peer` and `rep` and allow modifications to the list. The `rd` modifier for the type argument makes its ownership modifier irrelevant but restricts it to instances of `Data` (and not a subtype).

Before going into detail, let us have a look at the practicality. First, for the readable modifier `rd`. It can solely be used in type arguments as it makes no sense as a main modifier — `any` works just as fine there. It is applied in situations where we need an exact type as a type argument such as `List<Data>`. But how often do these situations occur? We are convinced that such cases are rare. Hardly ever, we want to restrict the set of possible types to a particular one. We believe that in the vast majority of these cases, the programmer actually meant to use a wildcard such as `List<? extends Data>`, or `List<? super Data>`. The reason why it occurs so often in code is because only experienced Java programmers are familiar with wildcards, or know that generic types are invariant. We think in most cases a normal wildcard is preferred to a universe wildcard.

Furthermore, we are convinced that in every case a `wr` modifier would be useful, there is something wrong with the ownership structure. A writable modifier means that we do not care whether we own the object, or it is peer to the current object. We invested quite some time trying to find a reasonable example where this would be useful, but we failed to come up with one. An object we own has another purpose than one of our peers and therefore should be used differently. Introducing a writable modifier supports bad practice which is very contradictory to the goal of ownership.

If we were to implement universe wildcards, they would add enormous complexity to Generic Universe Types. It would be necessary to apply an algorithm similar to capture conversion to universe wildcards. Unlike the existing ownership modifiers where an `any` modifier is equal to every other `any` modifier, the rules for universe wildcards were much more complex and we had to distinguish between different universe wildcards.

Based on the presented reasons, we have come to the conclusion to not pursue the idea of the writable modifier any further. For the readable modifier, we have found it to be useful for viewpoint adaptation. For example, when `rep` is combined with `peer List<rep Object>`, viewpoint adaptation yields `any List<any Object>`. The reason is that, although we know the context of the list's elements, we cannot express it with an ownership modifier — which is why we take `any`. And since `any` is not the exact description, we have to change the main modifier to `any` to prevent modifications to the list and preserve type safety.

If we extend the definition of the readable modifier to denote either a specific context — not just `peer` or `rep` but arbitrary combinations of the two — or any context, we can change viewpoint adaptation to yield `rep List<rd Data>` for the example above. That allows modifications to the list such as deletions of objects or rearranging them, but not insertions. This idea is described by Daniel Schregenberger in his Master's thesis [24]. He calls the modifier `unknown`.

Question is why the presented viewpoint adaptation is correct. We compare it to wildcards. Every time a type containing a wildcard is read, capture conversion is applied. This also happens to universe wildcards: When a universe wildcard occurs in a type that is read, the universe wildcard gets bound and assignments to it are prohibited. Imagine we wanted to insert an element into the list. We would call method `put` with signature `void put(T e)`. Substituting the type variable by its type argument yields `void put(rd Data e)`. `rd` is already bound since the access happens on the list and therefore, calling the method is impossible with an argument different from `null`.

The similarity of wildcards and the `rd` modifier is also true for subtyping and therefore for assignability. There are two possible relationships: Either `any` is a subtype of `rd`, or vice versa. In the first case, `any` is assignable to `rd` but `rd` is not assignable to `any`. This is wrong as `any` can refer to every context. In the second case, `rd` is assignable to `any` but `any` is not assignable to `rd`. This is wrong as well because `rd` stands for each of the modifiers and thus also for `any`.

By looking at `rd` as the wildcard version of `any`, we can describe the subtype relationship. We compare it to types `Data` and `? extends Data`. `rd` is assignable to `any` and so is a reference of type `? extends Data` to a reference of type `Data`[5]. Vice versa, `any` is assignable to an unbound `rd` and so is type argument `Data` assignable to type argument `? extends Data`.

[5]Although wildcards must only occur as type arguments, it is possible to get such types. For example, if we have a `List<? extends Data>` and call method `get`, its return type then is a type variable created by capture conversion from `? extends Data`.

Chapter 4

Static Universe Type Inference

In this chapter, we give an overview of the inference process in Section 4.2. Based on the data model described in Section 4.3, we present the algorithms for all kinds of constraints in Section 4.4. The existing tool and how to extend it are outlined in Section 4.5.

4.1 Introduction

Type inference is the ability to deduce automatically the type of a variable. It is often a characteristic of functional programming languages, but it can also be applied to object-oriented programming languages, e.g. it is planned for C# 3.0. The ability to infer types automatically makes many programming tasks easier, leaving the programmer free to omit type annotations while maintaining type safety.

Universe type inference refers to an automatic deduction of the ownership modifier of a type, i.e. its goal is not to deduce the entire type of a variable but the ownership modifier. As opposed to defaulting explained in Section 2.9 which simply puts all objects in the root context, universe type inference takes a more sophisticated approach and tries to create a deep ownership structure.

There are two inherently different approaches to universe type inference:

- Runtime universe type inference
- Static universe type inference

Runtime universe type inference executes the Java program on the Java virtual machine with a tracing agent attached to it. It then takes the trace file which contains runtime information like object creations, field accesses and method calls and builds a data structure representing this information. Based on that, the object store gets partitioned in context and valid ownership annotations are determined. Finally, these annotations are written to an XML annotation file.

Static universe type inference looks at the static information, i.e. it makes use of the source code to deduce ownership modifiers without executing the program. The source code is parsed and constraints are generated from the abstract syntax tree. Based on these constraints, a solution has to be found which is then written to an XML annotation file.

These two approaches are quite different and thus, have different qualities. The solution quality of the runtime approach heavily depends on the current run and therefore the code coverage whereas the static approach has a complete view of the program. Furthermore, the static inferrer can annotate method bodies and takes existing annotations into account which is not possible for the runtime inferrer because the interface through which it interacts with the Java virtual machine is limited.

For the Universe Type System, a lot of work was done looking into universe type inference. The projects of Frank Lyner [19] and Marco Bär [7] developed and applied the runtime approach whereas Nathalie Kellenberger [16] and Matthias Niklaus [21] developed tools for the static approach. Andreas Fürer [12] combined the two approaches by taking the best from each and packing them in several Eclipse [5] plugins.

In this chapter, we will present how the static approach can be extended to cope with Generic Universe Types and thus also deduce ownership modifiers for type arguments. We limit ourselves to the core of Generic Universe Types as it is presented in [9], e.g. we will not deal with arrays and wildcards. We will build on Matthias Niklaus' project which has been integrated into Andreas Fürer's inference pack.

We decided to go with the static approach since there is no built-in support for generic types at runtime in Java which makes the runtime approach not applicable. Once Mathias Ottiger's work on adding runtime support for Generic Universe Types [23] is done, dynamic inference is feasible as well.

4.2 Inference Process

In this section, we are going to explain how the inference process works on a high level. The description is based on the static universe inference tool developed by Mathias Niklaus [21]. Although the tool is designed for the Universe Type System, the process will remain the same for Generic Universe Types. We assume the tool already handles Generic Universe Types, i.e. this is how the process will work once static universe inference has been extended to cope with Generic Universe Types.

Once it has been decided to use Generic Universe Types for an existing project, it can be quite time-consuming to annotate the code. The project is already valid universe code as defaulting puts all objects in the root context. For ownership to add static guarantees, we have to create a deep hierarchy. This can either be done by hand, or we use an universe inference tool that gives us a first solution which can then be refined if necessary.

At the beginning, there is normal Java code such as the sample code presented in Listing 4.1. We will use this code throughout the section to show how the static universe inference tool processes it. Additionally, it is also possible to pass partially annotated code, i.e. code containing some modifiers. These are either treated as fixed, or as the preferred ownership modifier for the corresponding position.

Listing 4.1: Normal Java code to be annotated by the static universe inference tool

```
class Container<T extends Data> {
        T element;

        T get() {
                return element;
        }

        void set(T element) {
                this.element = element;
        }
}

class StaticInference {
        Container<ExtendedData> container;

        void main() {
                container = new Container<ExtendedData>();
                container.set(new ExtendedData());
        }
}
```

The inference process is started by passing the source code to the inference tool. Thanks to Andreas Fürer's inference pack [12], there now is an Eclipse plug-in that provides you with a simple user interface where you can specify the code you would like to have inferred.

As a first step, the code is parsed and type checked by the JML tools [18] with universe checks turned off, i.e. Java type rules are enforced and ownership modifiers are ignored. In order to allow partially annotated programs, ownership modifiers are permitted in the syntax . This way, they are stored in the abstract syntax tree and can be accessed afterwards.

Before byte code is written, the static inferrer intercepts. It takes over and traverses the entire abstract syntax tree to determine all positions where an ownership modifier has to be placed and therefore inferred. At each of these positions, it creates a new so-called *variable* that can represent any of the three possible ownership modifiers `peer`, `rep` and `any`.

While traversing the abstract syntax tree, the inferrer also creates so-called *constraints*. The idea is to create restrictions for variables that model the type rules of Generic Universe Types. A valid solution is then an assignment of one ownership modifier to each variable that satisfy all constraints. By satisfying all constraints, the type rules of Generic Universe Types are fulfilled and we have found a valid ownership annotation for the given program.

Constraints are recorded in a data structure. They represent the core of the inference process as they model exactly all rules of Generic Universe Types. For that purpose, there exist different kinds of constraints which we will introduce in Section 4.4. In the example below, we present you with a few examples to give you a basic idea how they look.

Listing 4.2: The code with the ownership modifiers to be inferred made explicit

```
class Container<T extends u1 Data> {
        T element;

        T get() {
                return element;
        }

        void set(T element) {
                this.element = element;
        }
}

class StaticInference {
        u2 Container<u3 ExtendedData> container;

        void main() {
                container = new u4 Container<u5 ExtendedData>();
                container.set(new u6 ExtendedData());
        }
}
```

In Listing 4.2, we have taken the input of the inference tool and placed a new variable at each position where an ownership modifier has to be inferred. The inference tool does not place them explicitly in the source code. We have numbered the variables and list them to be able to give some examples of constraints that are built while traversing the abstract syntax tree[1]:

- We have learned that `rep` must not occur in upper bounds of class type variables. Therefore, we introduce a constraint for `u1` that it must not be `rep`. This kind of constraint is called a *declaration constraint*.

[1]This list is not complete. There are more constraints.

- Another declaration constraint can be found on line 18. There we call non-pure method `set` on field `container`. For this call to be valid, the main modifier of `container`'s type must not be `any`. We introduce this constraint for `u2`.
- On line 17, a new instance of `Container` is created and assigned to field `Container`. This assignment is only valid if the newly created object's type, `u4 Container<u5 ExtendedData>`, is a subtype of `container`'s type, `u2 Container<u3 ExtendedData>`. In this case, several ownership modifiers are involved which makes it more complicated. On a higher level, we can say that there is a *subtype constraint* between these two types.

After the abstract syntax tree has been traversed and all constraints have been stored in a data structure, they have to be solved in order to find a solution. The existing static universe inference tool uses a MAX-SAT solver for that purpose, i.e. being passed a number of clauses in *conjunctive normal form (CNF)* it tries to find an assignment that satisfies as many clauses as possible.

To make use of a MAX-SAT solver, the constraints have to be translated into boolean formulas in CNF first. Each variable is encoded with a fixed number of bits. Then, each constraint is mapped to boolean formulas in CNF by pre-determined schemas which exist for each kind of constraint. The output, i.e. the clauses, are written to a file and passed to the solver.

Based on the complexity of the satisfiability problem, it might take some time until the solver comes back with a solution. The solution is read by the inference tool which converts it to an assignment for the variables. This is then presented to the developer. Below you find a possible ownership annotation for the example.

Listing 4.3: A possible solution to the example presented above

```
 1  class Container<T extends any Data> {
 2          T element;
 3
 4          T get() {
 5                  return element;
 6          }
 7
 8          void set(T element) {
 9                  this.element = element;
10          }
11  }
12
13  class StaticInference {
14          rep Container<rep ExtendedData> container;
15
16          void main() {
17                  container = new rep Container<rep ExtendedData>();
18                  container.set(new rep ExtendedData());
19          }
20  }
```

As you can see from Listing 4.3, all constraints mentioned above are satisfied[2]: No `rep` occurs in the upper bound of type variable `T`, `container`'s main modifier is not `any` and the assignment on line 18 is correctly typed.

What we have just presented is not the only valid solution. Each `rep` modifier could as well be replaced by `peer` and the annotation would still be valid. So, why was `rep` preferred to `peer`? The inferrer tries to create a deep ownership structure. This is achieved by preferring `rep` to the other two modifiers as the only way to create such an ownership structure is to have as many `rep` modifiers as possible — especially in object instantiations.

[2]As there is no implementation of a static inferrer for Generic Universe Types yet, this solution has been created by hand.

The solver's weighting feature is used to specify the distribution of the ownership modifiers. This feature allows to specify a weight for every bit for which the solver has to find a boolean value. We use it to prefer the bit setting the ownership modifier to `rep`. The developer can freely change this default behavior. The tool allows to set a weight for every ownership modifier depending on where it occurs, e.g. in a field or a local variable. Note that preferences are only taken into account when more than one ownership modifier is applicable at a certain position. Otherwise, the only option is taken in order to get a valid solution.

Another important aspect of the inference process is the developer's influence. Once an initial solution is presented, the developer can freely change any annotations and thereby give input. The inferrer then tries to take the developer's choices into account. It integrates the input into the constraints and passes them again to the solver. That way, the annotation can be refined step-by-step until the desired solution is reached.

The developer's most important task when giving input is specifying pure methods. The inferrer does not look at the methods and figure out which methods are pure and which are not. It treats every method without `pure` annotation as non-pure.

As the inferrer's last step, the final inferred annotation is written to an XML file. The annotations XML file representing the solution presented in Listing 4.3 is given in Listing 4.4. The XML file contains all necessary information to have a tool insert the annotations into the source code. The tool doing this job was developed by Andreas Fürer [12]. It is also included in the inference pack allowing a smooth integration of the ownership modifiers in the code.

Listing 4.4: The annotations XML file written by the static inferrer

```
<?xml version="1.0"?>
<annotations>
    <head>
        <target>java</target>
        <style>types</style>
        <comment>Created by hand</comment>
    </head>
    <class name="Container">
        <type_variable index="0" name="T" line="1">
            <upper_bound index="0" modifier="any" class="java.lang.Object"/>
        </type-variable>
    </class>
    <class name="StaticInference">
        <field name="container" line="14">
            <type modifier="rep" class="Container">
                <type_argument index="0" modifier="rep" class="ExtendedData"/>
            </type>
        </field>
        <method name="main" line="16">
            <new index="0" modifier="rep" class="Container">
                <type_argument index="0" modifier="rep" class="ExtendedData"/>
            </new>
            <new index="1" modifier="rep" class="ExtendedData"/>
        </method>
    </class>
</annotations>
```

4.3 Data Model

Before we will explain constraints in detail and how they are generated, we introduce the data model used to represent types, variables and constraints. The algorithms for constraint generation will work on this data model. An UML diagram can be found in Figure 4.1.

Unlike the Universe Type System where one type equals one ownership modifier, a type in Generic Universe Types can have an arbitrary number of ownership modifiers. Thus, our data model has two basic components: types and ownership modifiers.

A *Variable* — a term we adopted from [21] — stands for a single ownership modifier. It can either be `any`, `peer` or `rep`. A new instance is created for every ownership modifier to be inferred. Each *Variable* always belongs to a type.

A *Type* represents any kind of type, be it the type of a field, the return type of a method, the type of an expression or a type variable, that are relevant to build constraints. We do not model types like primitive types or `null` as they are not needed to model constraints. *Type* has two subclasses as they are defined in [9]: the *Non-variable Type* and the *Type Variable*.

A *Non-variable Type* consists of three parts: an ownership modifier, a class and a list of type arguments. The ownership modifier is a reference to a *Variable*, i.e. each *Non-variable Type* has one *Variable* associated with it. The type arguments are a possibly empty list of *Types*.

A *Type Variable* is created for each type parameter of a generic class. It has an upper bound which is a *Type*, i.e. its upper bound could again be a *Type Variable*. Whenever a reference to the type variable is encountered in a type, the *Type Variable* is directly referenced in the data structure.

When generating constraints, the *Type Variable* is substituted by its upper bound or its type argument, whatever is appropriate. The *Type Variable* itself has no constraints since there is no *Variable* associated with it and therefore no ownership modifier has to be inferred. Constraints are associated with the upper bound or the type argument instead.

An instance of *Class Information* is created for each class encountered in the source code. It basically stores all the types that exist once per class. This means it contains the list of all *Type Variables* declared by this class and the type for `this`.

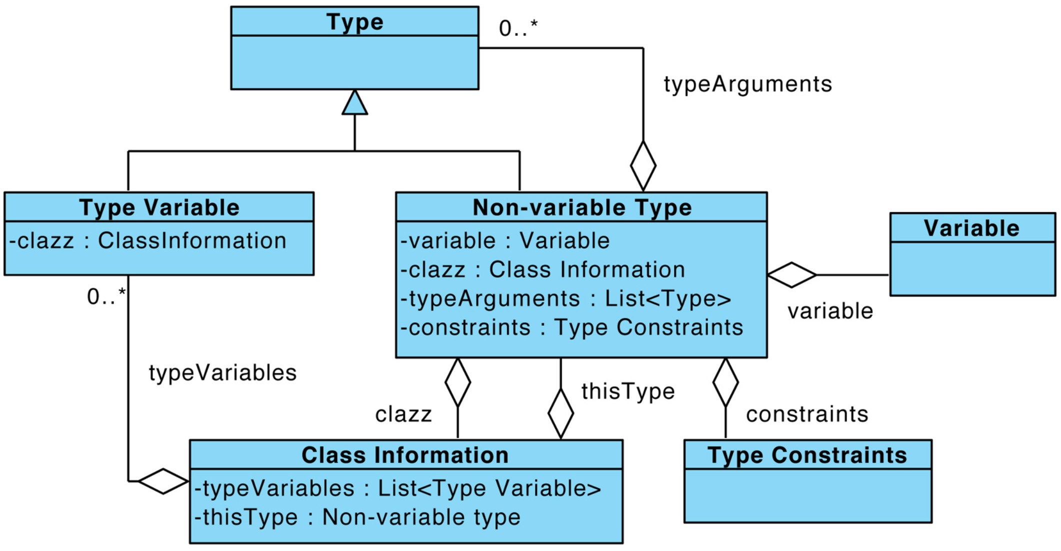
Type
0..*
typeArguments
Type Variable
-clazz : ClassInformation
Non-variable Type
-variable : Variable
-clazz : Class Information
-typeArguments : List<Type>
-constraints : Type Constraints
Variable
variable
0..*
typeVariables
clazz
thisType
constraints
Class Information
-typeVariables : List<Type Variable>
-thisType : Non-variable type
Type Constraints

Figure 4.1: UML diagram of the data model

Listing 4.5: An example to demonstrate how the data model represents different types

```
class InferTypeRepresentation<X> {
        X field;
3
4        void main() {
5                Object o = new LinkedList<Data>();
6                o = field;
7        }
8  }
```

To illustrate how the presented data structure stores various types, we will explain it for three examples shown in Listing 4.5. The generic class `Container` is already known from previous examples.

Type `LinkedList<Data>` on line 5 is stored as a *Non-variable Type*. It has a new *Variable*, i.e. an ownership modifier that has to be inferred, links to *Class Information* for `LinkedList` and has one type argument in its list. The type argument is a *Non-variable Type* and has yet another new *Variable*, links to `Data` and contains no type arguments.

The type for `field` on line 6 is represented by a *Type Variable* which got created when the declaration of type variable `X` was parsed. The type variable, since it is unbounded, was assigned a new *Non-variable Type* as its upper bound with a new *Variable*. It references *Class Information* for `Object` and has no type arguments.

The data model presented so far allows us to represent all relevant type information and to relate ownership modifiers to types. Yet, on a higher level, constraints are built from relationships of types to each other which then, on a lower level, are mapped to variables relating to each other. Examples for relationships of types are the combination of a type A with type B that results in a type C or a type A has to be a subtype of B. Such relationships are stored in *Type Constraints*.

Each *Non-variable Type* has an instance of *Type Constraints* associated with it that maintains all constraints encountered so far in the source code. *Type variables* do not have constraints as they do not have any ownership modifiers. Their constraints are attached to the upper bound. The different kinds of constraints presented in the following are all stored separately in *Type Constraints*.

Note that we have only presented the core of the data model necessary to completely model the constraints for Generic Universe Types. For example, a class dealing with methods, i.e. their type variables, parameter types and return type, has to be added.

4.4 Constraints

As we have explained in Section 4.2, constraints are restrictions modeling the rules of Generic Universe Types. On the level of the data model, we store the relationships between each *Non-variable Type* and an arbitrary number of *Type*s. These are then mapped to dependencies of *Variable*s which are eventually written to a file as boolean formulas.

You can see from this description, that there are basically two steps to generating constraints: First, they are built in the data model. Then, in a second step, they are mapped to variables and written to a file. We perform these steps in two separate passes:

1. While traversing the abstract syntax tree which was created by the compiler, we extract all constraints and store them in *Type Constraints*. This includes creating *Type*s, *Variable*s, and for each class, *Class Information*. This first phase is called *constraint building*.

2. Once this is done, we no longer need to look at the abstract syntax tree. We now have all necessary information in our own data model which is tailored to our needs. Therefore, we look at the *Type Constraints* of every *Non-variable Type* and map them to boolean formulas. This second phase is called *constraint writing*.

In the following, we will introduce five kinds of constraints that allow us to model the type rules of Generic Universe Types[3]. For each kind of constraint, we present two algorithms:

[3]For those familiar with [9], we include well-formedness in the type rules as far as it is not already covered by the Java type rules. These have already been checked once we get to building constraints.

- The constraint building algorithm decides if this kind of constraint is built for the given input, i.e. we first list situations where a certain kind of constraint is applicable and the algorithm then decides if it is necessary for the given *Type*s. When the algorithm determines that a constraint is to be built, the constraint gets stored in the appropriate instance of *Type Constraints*.

- The constraint writing algorithm is executed for each constraint built, i.e. for each constraint that has been stored in some instance of *Type Constraints* by the corresponding constraint building algorithm. The constraint writing algorithm takes the relationships between *Type*s and maps them to boolean formulas creating relationships between *Variable*s which are then written to a file.

Note that we do not show the boolean formulas on a bit level. Instead, we use the *Variable*s and expressions like $u = peer$ to state that u is `peer`. We refer to [21] for various ways to encode ownership modifiers and therefore the *Variable*s. In some cases, we introduce new literals, i.e. bits, that are needed in the formulas. These can be mapped directly to the bit level.

4.4.1 Declaration Constraint

The declaration constraint disallows a certain ownership modifier. Unlike all other constraints, this kind of constraint is not only applicable to a *Non-variable Type* but also to a *Variable*. Applying it to a *Non-variable Type* means applying it to every *Variable* being part of the type, i.e. the given ownership modifier must not occur in any type argument either.

The declaration constraint is also used to fix a *Variable* to a certain ownership modifier. In that case, two declaration constraints are applied to prevent the other two ownership modifiers. The remaining ownership modifier is then the only valid solution.

We apply the declaration constraint in the following cases:

- formal parameter types of pure methods must only contain the `any` modifier
- upper bounds of class type variables must not contain the `rep` modifier
- the main modifier of the receiver type of a non-pure method call must not be the `any` modifier
- the main modifier of the receiver type of a field update must not be the `any` modifier
- if a field update does not happen on `this`, the field's declared type must not contain the `rep` modifier
- if the receiver of a non-pure method is different from `this`, none of its formal parameter types must contain the `rep` modifier
- the main modifier of an object instantiation must not be `any`
- the main modifier for boxing types such as `Integer` and `String` must be the `any` modifier

Building a Declaration Constraint

For all cases being listed above, declaration constraints have to be built. Compared to other kinds of constraints we will present later in this section, keeping track of declaration constraints is easy: It suffices to store for each ownership modifier whether it is allowed or not. This is done for every *Variable*. At creation time of the *Variable*, all ownership modifiers are allowed. A certain ownership modifier can then be prevented by applying a declaration constraint.

Listing 4.6: Sample code to demonstrate how declaration constraints are built

```
1 class DeclarationConstraintBuilder<X extends u1 Data> {
2         u2 Container<u3 Data> container;
3 
4         void main() {
5                 container = new u4 Container<u5 Data>();
6                 container.set(null);
7         }
8 }
```

In Listing 4.6 we have three cases requiring a declaration constraint:

- On line 1, `X`'s upper bound `u1 Data` must not contain the `rep` modifier. We apply a declaration constraint to a *Non-variable type* which means applying it to every *Variable* being part of it and disallow the `rep` modifier for all its *Variables* (here only `u1`).
- On line 5, we prohibit the `any` modifier for *Variable* `u4` as new objects have to be in a specific context. To this end, we use a declaration constraint for *Variable* `u4`. Since it does not apply to the entire type, `any` is still a valid ownership modifier for `u5`.
- On line 6, non-pure method `set` is called on field `container`. This must not happen on `any` references. Therefore, we use a declaration constraint on *Variable* `u2` to prohibit the `any` modifier.

Writing a Declaration Constraint

To write the declaration constraints, we use the following algorithm for each *Variable*:

Writing a Declaration Constraint

Input:	A *Variable* u
Output:	Up to two clauses in CNF representing the declaration constraints of *Variable* u
Side-effect:	-

1. If *Variable* u has a declaration constraint prohibiting ownership modifier `peer`, the following clause is added: $\neg u = peer$
2. If *Variable* u has a declaration constraint prohibiting ownership modifier `rep`, the following clause is added: $\neg u = rep$
3. If *Variable* u has a declaration constraint prohibiting ownership modifier `any`, the following clause is added: $\neg u = any$

Note that at most two ownership modifiers must be forbidden. If all three are forbidden, we issue an error.

We omit to give the clauses for the example as it is straightforward.

4.4.2 Combination Constraint

A combination constraint expresses that the result of the combination of a *Type* A with a *Type* B is *Type* C, i.e. adapting *Type* B to the viewpoint A results in *Type* C.

A combination constraint is needed where viewpoint adaption is required, which is in the following cases:

- for object creations, the new object's type has to be combined with each type variable's upper bound
- for method calls, the receiving object's type has to be combined with each method parameter's type and each method type variable's upper bound
- for expressions, all occurring fields and return types of methods have to be combined from left to right

Internal Types

Whenever we combine two *Types*, a new third *Type* gets created whose ownership modifiers are not visible to the outside, i.e. they will not annotate any type in the source code, but they are needed internally to relate the two original *Types* to other types. Let us consider the following example:

Listing 4.7: Sample code to show why internal types are required for constraints

```
1 class InternalTypes {
2         u1 InternalTypes field;
3 
4          void main() {
5                 u2 InternalTypes iTypes = new u3 InternalTypes();
6                 u4 InternalType result = iTypes.field;
7         }
8 }
```

In Listing 4.7 on line 5, we create a new instance of `InternalTypes` and assign it to a variable `iTypes`. We then access `iTypes`'s `field` and assign it to another variable `result`. There is no immediate relationship between the types of `field`, `iTypes` and `result`, but line 6 imposes a constraint: The combination of `iTypes` type with `field`'s type has to be assignable to `result`'s type. In order to simplify this, we create an internal *Type* representing the combination of `iTypes` *Type* with `field`'s *Type* which then has to be assignable to `result`[4]. To reduce complexity of the resulting constraints, this *Type* is reused for each combination of these two *Types*.

[4]Assignability is ensured by a subtype constraint which is presented in section 4.4.3.

Building a Combination Constraint

Building a combination constraint follows a process which we describe in the form of an algorithm. The input of the algorithm consists of the left type of the combination for A and the right type of the combination of B.

Building a Combination Constraint

Input:	Two *Type*s A and B
Output:	A *Type* being the combination of *Type* A with *Type* B
Side-effect:	A combination constraint is added if applicable

1. If A is a *Type Variable*, we take its upper bound instead. We do this until we reach a *Non-variable Type*. This then becomes type A.
2. If B is a *Type Variable*, we return the type argument for it. The type argument can be accessed through *Non-variable Type* A. No constraint is created. We are done.
3. Otherwise, we check if A has already been combined with B before. We do a lookup and if this combination already exists, we return that *Type*. We are done.
4. Otherwise, we look at *Non-variable Type* B. We create a new *Non-variable Type* C with a new *Variable*. It links to the same *Class Information* as B. The type arguments of the newly created *Non-variable Type* is a new list that is created by applying the following to each of B's type arguments:
   - If the type argument is a *Non-variable Type*, we again create a new *Non-variable Type* with a new *Variable*. It links to the same *Class Information* as the original type argument and we recursively proceed with each of its type arguments.
   - If the type argument is a *Type Variable*, we take its type argument which can be found as part of *Non-variable Type* A.
5. We add *Non-variable Type* C from the previous step as the result of the combination of A with B as a pair consisting of B and C to A's *Type Constraints*. We return C.

As you can see, the newly created *Non-variable Type* C is a copy of B, except that it has new *Variable*s and all *Type Variable*s are substituted by their type argument. It is an internal *Type*, i.e. its *Variable*s will not be visible in the program to be annotated. Instead, it is used in other constraints.

There are cases when no combination constraint is built: When the constraint already exists or when no viewpoint adaptation takes place since the type argument is returned. Furthermore, if a type cannot be represented as an instance of *Type*, e.g. if it is a primitive type or `this`, then no constraint is built either because no viewpoint adaptation is required.

The keyword `this` can have two meanings in our context: First, it indicates that the next expression refers to a member of the current type. In this case, it is not relevant to us as no viewpoint adaptation takes place and the compiler has already figured out which reference to use, e.g. the field is hidden by a homonymous local variable. Second, it is a reference to the current type with no following member in which case we pull the *Type* from the *Class Information*.

Listing 4.8: An example to explain the combination constraint algorithms

```
class Data {
	u1 Object property;

 4     u2 Object getProperty() {
 5         return property;
 6     }
 7
 8     void setProperty(u3 Object property) {
 9         this.property = property;
10     }
11 }
12
13 class ExtendedData extends Data {
14 }
15
16 class GenericHelper<X extends u4 Object, Y extends u5 Object> {
17 }
18
19 class CombinationConstraint<X extends u6 Data> {
20     X field;
21     u7 GenericHelper<u8 Object, X> helper;
22
23     void main() {
24         u9 CombinationConstraint<u10 ExtendedData> constraint;
25
26         // initialization omitted
27
28         u11 Object o = field.getProperty();
29         o = constraint.helper;
30     }
31 }
```

We will use the example in Listing 4.8 to show you the application of the combination constraint. Class `GenericHelper` only has two type variables `X` and `Y`. Classes `Data` and `ExtendedData` are listed explicitly in this example as we access a field for demonstration purposes which involves their ownership modifiers in the constraints.

Let us have a look at line 28 first: `field.getProperty()` requires a combination constraint as we need the resulting type to check assignability to local variable `o`. Input to the algorithm is *Type Variable* `X` as *Type A* and *Non-variable Type* `u2 Object` (the return type of method `getProperty`) as *Type B*. We will go through it, step-by-step:

1. Since *Type A*, `X`, is a *Type Variable*, we take its upper bound, `u6 Data`, as the new *Type A*. This is a *Non-variable Type*, so we are fine.
2. *Type B*, `u2 Object`, is not a *Type Variable*. So, we are done with this step.
3. There has not been an occurrence of this combination before. We go on to the next step.
4. We create a new *Non-variable Type* which has a new *Variable* `u12` and also links to *Class Information* `Object`. There are no type arguments. This is *Type C*.
5. We add an entry to *Type A*'s (which is `X`'s upper bound `u6 Data`) *Type Constraints* that the newly created type results from combining *Type A* with *Type B*. *Type C* is returned. It can then be used to add a constraint that ensures its assignability to `o`'s *Type*.

It is important to see that, although the return type of method `getProperty` of class `Data` and the local variable `o` of method `main` are both of type `Object`, as well as the newly created type, they do not refer to the same *Non-variable Type*. Each occurrence of a type has a separate instance of a *Non-variable Type* and therefore a different *Variable*. *Type*s referring to the same class do share the same instance of *Class Information* which only exists once per class.

Let us now do the same for line 29. This time, *Type A* is *Non-variable Type* `u9 CombinationConstraint<u10 ExtendedData>` and *Type B* is *Non-variable Type* `u7 GenericHelper<u8 Object, X>`.

1. *Type A* is not a *type variable*. Skip.
2. *Type B* is not a *type variable*. Skip.
3. These two *Type*s have not been combined before. Skip.
4. We create a new *Non-variable Type* with a new *Variable* `u13` which links to *Class Information* `GenericHelper`. It has two type arguments: Another new *Non-variable Type* with a new *Variable* `u14` that links to `Object`. The other type argument is a *Type Variable* and we therefore simply replace it by its type argument `u10 ExtendedData`. You can see here that the type argument can be found in *Type A* and that no new *Variable* is created. The resulting *Type C* is `u13 GenericHelper<u14 Object, u10 ExtendedData>`.
5. This newly created *Type C* is added as the result of the combination with `u7 GenericHelper<u8 Object, X>` to the *Type Constraints* of *Type* `u9 CombinationConstraint<u10 ExtendedData>`.

Writing a Combination Constraint

For each combination constraint, we use the algorithm stated below taking three *Non-variable Type*s A, B and C to write the combination constraint. As input, we have for A the *Non-variable Type* the *Type Constraints* belongs to, for B the *Non-variable Type* A is combined with, and for C the resulting *Non-variable Type* from the combination constraint building algorithm.

Writing a Combination Constraint

Input:	Three *Non-variable Type*s A, B and C
Output:	An arbitrary number of clauses in CNF representing the combination constraint of *Type A* combined with *Type B* resulting in *Type C*
Side-effect:	-

1. From A, take the main modifier. We call it u.
2. From B, take the main modifier. We call it u'. Furthermore, take all ownership modifiers occurring in the transitive closure B's type arguments. We call them u_1 $\dots u_n$.
3. From C, take the modifier corresponding to u'. We call it v.
4. Write the following clauses:

$$\begin{array}{ll} u = any \rightarrow v = any & \wedge \\ u' = any \rightarrow v = any & \wedge \\ u' = rep \rightarrow v = any & \wedge \\ u = peer \wedge u' = peer \rightarrow v = peer \vee r & \wedge \\ u = rep \wedge u' = peer \rightarrow v = rep \vee r & \wedge \\ r \rightarrow v = any & \wedge \\ r \leftrightarrow u_1 = rep \vee ... \vee u_n = rep & \end{array}$$

5. Repeat steps 2 - 5 (recursively) for each of B's type arguments. Thereby, replace B by the current type argument. If the type argument is a *Type Variable*, nothing has to be done.

Simply spoken, A's main modifier is combined with each modifier of B. The result of each of these combinations is assigned to C's corresponding modifier. We know that C is a copy of B. The clauses written in step 4 model the type combinator as follows:

- If `any` occurs, the result is again `any` (clauses 1 and 2)
- If the second modifier is `rep`, the result is `any`[5] (clause 3)
- `peer` (`rep`) combined with `peer` is again `peer` (`rep`) unless one of the type arguments contains `rep` (clauses 4 and 5)
- If `rep` occurs in the type arguments, the result is `any` (clause 6)
- r states if `rep` is contained in the transitive closure of the second *Type*'s type arguments (clause 7)

r is a new bit that is introduced every time step 4 is executed, i.e. multiple new bits could be created during the execution of the algorithm. It tells us if we have to change the ownership modifier to `any`. If we have no *Variables* in the type arguments or no type arguments, r is *false* by default.

We go back to the example in Listing 4.8 and take this one step further by writing the clauses. We will use the combination on line 29. There we combined `u9 CombinationConstraint<u10 ExtendedData>` with `u7 GenericHelper<u8 Object, X>`. We have already created the resulting *Non-variable Type* which is `u13 GenericHelper<u14 Object, u10 ExtendedData>`.

With the help of our algorithm, we will relate the *Variables*. The input for our algorithm is `u9 CombinationConstraint<u10 ExtendedData>` for A, `u7 GenericHelper<u8 Object, X>` for B and `u13 GenericHelper<u14 Object, u10 ExtendedData>` for C.

1. u = `u9`
2. u' = `u7`, u_1 = `u8`
3. v = `u13`
4. The following clauses are written:

$$\begin{array}{ll} \texttt{u9}= \mathit{any} \rightarrow \texttt{u13}= \mathit{any} & \wedge \\ \texttt{u7}= \mathit{any} \rightarrow \texttt{u13}= \mathit{any} & \wedge \\ \texttt{u7}= \mathit{rep} \rightarrow \texttt{u13}= \mathit{any} & \wedge \\ \texttt{u9}= \mathit{peer} \wedge \texttt{u7}= \mathit{peer} \rightarrow \texttt{u13}= \mathit{peer} \vee r_1 & \wedge \\ \texttt{u9}= \mathit{rep} \wedge \texttt{u7}= \mathit{peer} \rightarrow \texttt{u13}= \mathit{rep} \vee r_1 & \wedge \\ r_1 \rightarrow \texttt{u13}= \mathit{any} & \wedge \\ r_1 \leftrightarrow \texttt{u8}= \mathit{rep} & \end{array}$$

5. We repeat steps 2 - 5 for B's type arguments `u8 Object` and `X`. Note that there is nothing to be done for `X` as it is a *Type Variable*.

2. We now replace *Non-variable Type B* by the current type argument. This is `u8 Object`. u' = `u8` and no u_i.

3. The corresponding v to `u7` is `u13`.

4. These clauses are written:

$$\begin{array}{ll} \texttt{u9}= \mathit{any} \rightarrow \texttt{u14}= \mathit{any} & \wedge \\ \texttt{u8}= \mathit{any} \rightarrow \texttt{u14}= \mathit{any} & \wedge \\ \texttt{u8}= \mathit{rep} \rightarrow \texttt{u14}= \mathit{any} & \wedge \\ \texttt{u9}= \mathit{peer} \wedge \texttt{u8}= \mathit{peer} \rightarrow \texttt{u14}= \mathit{peer} & \wedge \\ \texttt{u9}= \mathit{rep} \wedge \texttt{u8}= \mathit{peer} \rightarrow \texttt{u14}= \mathit{rep} & \end{array}$$

As you can see, the clauses become simpler as there is no r to consider.

[5]Note that the `this` modifier is not modeled.

5. There are no type arguments.

4.4.3 Subtype Constraint

When *Non-variable Type B* is added to the list of subtype constraints of *Non-variable Type A*, it means that B has to be a subtype of A.

The subtype constraint is applied in the following cases:

- for assignments, the right hand side has to be a subtype of the left hand side
- for a method call, each argument's type has to be a subtype of the corresponding viewpoint adapted formal parameter's type
- for return statements, their type has to be a subtype of the method's return type
- for instantiations of type variables, their type arguments have to be a subtype of the corresponding upper bound

Building a Subtype Constraint

To enforce the subtype relationship, we can build on Java's subtyping rules because the entire program has already been type checked by a Java compiler before we start building the constraints. Thus, we do not have to care about whether classes are subtypes of each other but we only have to look at the ownership modifiers.

The following algorithm builds a constraint for two *Type*s A and B where B has to be a subtype of A:

Building a Subtype Constraint

Input:	Two *Type*s A and B
Output:	-
Side-effect:	A subtype constraint is added if applicable

1. If A is a *Type Variable*, we are done. No constraint is added.
2. Otherwise, if B is a *Type Variable*, we take its upper bound until we get a *Non-variable Type*. This type then becomes *Type B*.
3. We check if A's *Type Constraints* already contains B as a subtype. If that is the case, we are done.
4. Otherwise, we add *Non-variable Type B* to the list of subtype constraints of *Non-variable Type A*.

If the subtyping rules of Generic Universe Types for *Type*s A and B are stricter than in Java, we build a constraint. No constraint is required if A is *Type Variable* because then B is a *Type Variable* as well due to Java's subtyping rules. Or if B were `null`, for which we build no constraint either.

We will illustrate the algorithm by an example:

Listing 4.9: An example to explain the subtype constraint algorithms

```
class BaseClass<T extends u1 Data, U extends u2 Object> {
}

class SimpleClass<X extends u3 Object> extends BaseClass<u4 ExtendedData, X> {
}

```

```
 7
 8  class SubtypeConstraint<X extends u5 SimpleClass<u6 Data>, Y extends X> {
 9          X fieldX;
10          Y fieldY;
11
12          void m(u7 BaseClass<u8 ExtendedData, u9 Data> b) {
13          }
14
15           void main() {
16                  fieldX = fieldY;
17                  m(fieldY);
18          }
19  }
```

In Listing 4.9 we give two examples requiring a subtype constraint. On line 16, an assignment is made: A reference of *Type* `Y` is assigned a reference of *Type* `X`. We have to ensure that the expression's type of the right hand side of the equal sign is a subtype of the left hand side. Therefore, the two *Type*s function as input to the subtype constraint building algorithm: *Type Variable* `X` for A and *Type Variable* `Y` for B:

1. We are done since A is a *Type variable*.

The other occasion where a subtype constraint is applied is found on line 17: In case of a method call, the actual parameter's type has to be a subtype of the type of the method's formal parameter. We apply our algorithm with the formal parameter's *Non-variable Type* `u7 Container<u8 Data>` for A and the actual parameter's *Type Variable* `Y` for B:

1. Skip since A is not a *Type Variable*.
2. We take `Y`'s upper bound, `X`. Since it is again a *Type Variable*, we repeat this step and take `X`'s upper bound, *Non-variable Type* `u5 Container<u6 Data>`. This becomes B.
3. This constraint does not yet exist.
4. We add a subtype constraint to *Non-variable Type* A's list: *Non-variable Type* `u5 Container<u6 Data>` indirectly representing the upper bound of *Type Variable* `Y` has to be a subtype of *Non-variable Type* `u7 BaseClass<u8 ExtendedData, u9 Data>` representing the formal method parameter `b`'s type.

Writing a Subtype Constraint

Writing a subtype constraint comes down to relating the corresponding *Variables* of the two *Non-variable Types*. Basically, the two main modifiers have to be subtypes and the remaining modifiers, which occur in the type arguments, have to be equal. Additionally, limited covariance has to be taken into account.

For the algorithm below, the *Non-variable Type* to which the instance of *Type Constraints* belongs is A and the *Non-variable Type* retrieved from *Type Constraints* is B.

Writing a Subtype Constraint

Input:	Two *Non-variable Type*s A and B
Output:	An arbitrary number of clauses in CNF representing the subtype constraint making B a subtype of A
Side-effect:	-

1. If the *Class Information* of A and B refer to different classes, walk up B's supertype hierarchy and substitute the type arguments until A and B refer to the same class. This then becomes the new *Non-variable Type* B. It might be necessary to backtrack as there can be multiple supertypes due to interfaces.
2. From A, take the main modifier. We call it u_0.
3. From B, take the main modifier. We call it u'.
4. Write the following constraints:

$$\begin{array}{ll} u' = any \rightarrow u_0 = any & \wedge \\ u_0 = peer \rightarrow u' = peer & \wedge \\ u_0 = rep \rightarrow u' = rep & \end{array}$$

5. For each type argument of A: If the type argument is a *Type Variable*, go to the next type argument. Otherwise, if it is a *Non-variable Type*, take its ownership modifier and call it u_i where i is the current nesting level. Take the corresponding modifier of B and call it u'. Then write the following clauses:

$$\begin{array}{ll} u' = any \rightarrow u_i = any & \wedge \\ u_i = peer \rightarrow u' = peer & \wedge \\ u_i = rep \rightarrow u' = rep & \wedge \\ u_i = any \rightarrow u' = any \vee a & \wedge \\ u' = peer \rightarrow u_i = peer \vee a & \wedge \\ u' = rep \rightarrow u_i = rep \vee a & \wedge \\ a \leftrightarrow u_0 = any \wedge ... \wedge u_{i-1} = any & \end{array}$$

   This is done recursively for nested type arguments, i.e. continue with nested type arguments before moving to the next type argument on the same nesting level. Thereby, A is replaced by the current type argument.

This algorithm needs some explanation. Before we can look at the *Variables*, we have to have A and B refer to the same class. This is done in the first step by taking the subtype B, recursively traversing its supertype hierarchy and thereby substituting its type arguments. Using a depth first or breadth first search, B will eventually be a type of the same class as A.

In step 4, we create three clauses that ensure the subtype relationship of both main modifiers. If the subtype's modifier is `any`, the supertype's modifier has to be `any` as well (clause 1). If the supertype's modifier is `peer` (`rep`), the subtype's modifier also needs to be `peer` (`rep`) (clauses 2 and 3).

Since A and B refer to the same class due to step 1, they have the same number of type arguments. This is also true for nested type arguments due to Java's invariance for generic types. In step 5, we enforce equality of all corresponding *Variables* occurring in A and B. Where applicable, we allow limited covariance which is expressed by the a bit. It expresses if all outer modifiers are `any` which loosens the equality and only enforces a subtype relationship between the two *Variables*. This is done recursively for nested type arguments. We name the outer *Variable*s of A u_i where i is the level of nesting.

To give an example, we continue on the one presented in Listing 4.9. We have already built a subtype constraint and apply the subtype constraint writing algorithm to write the clauses. The input to the algorithm is the supertype `u7 BaseClass<u8 ExtendedData, u9 Data>` for A and the subtype `u5 SimpleClass<u6 Data>` for B.

1. A refers to `BaseClass` whereas B refers to `SimpleClass`. Therefore, we have to traverse `SimpleClass`'s supertype hierarchy. Its only supertype is `BaseClass<u4 ExtendedData, X>` where we substitute `X` by its argument `u6 Data`. So, we get for B `u7 BaseClass<u4 ExtendedData, u6 Data>`.

2. $u_0 = \texttt{u7}$

3. $u' = \texttt{u5}$

4. The following constraints are written:

   $\texttt{u5} = \textit{any} \rightarrow \texttt{u7} = \textit{any} \quad \wedge$
   $\texttt{u7} = \textit{peer} \rightarrow \texttt{u5} = \textit{peer} \quad \wedge$
   $\texttt{u7} = \textit{rep} \rightarrow \texttt{u5} = \textit{rep}$

5. We take A's first type argument `u8 ExtendedData`. $u_1 = \texttt{u8}$ and correspondingly $u' = \texttt{u4}$. The following clauses are written:

   $\texttt{u4} = \textit{any} \rightarrow \texttt{u8} = \textit{any} \quad \wedge$
   $\texttt{u8} = \textit{peer} \rightarrow \texttt{u4} = \textit{peer} \quad \wedge$
   $\texttt{u8} = \textit{rep} \rightarrow \texttt{u4} = \textit{rep} \quad \wedge$
   $\texttt{u8} = \textit{any} \rightarrow \texttt{u4} = \textit{any} \vee a \quad \wedge$
   $\texttt{u4} = \textit{peer} \rightarrow \texttt{u8} = \textit{peer} \vee a \quad \wedge$
   $\texttt{u4} = \textit{rep} \rightarrow \texttt{u8} = \textit{rep} \vee a \quad \wedge$
   $a \leftrightarrow \texttt{u7} = \textit{any}$

   Since `u8 ExtendedData` has no nested type arguments, we continue with the next argument on level $i = 1$ which is `u9 Data`. $u_1 = \texttt{u9}$ and correspondingly $u' = \texttt{u6}$. The following clauses are written:

   $\texttt{u6} = \textit{any} \rightarrow \texttt{u9} = \textit{any} \quad \wedge$
   $\texttt{u9} = \textit{peer} \rightarrow \texttt{u6} = \textit{peer} \quad \wedge$
   $\texttt{u9} = \textit{rep} \rightarrow \texttt{u6} = \textit{rep} \quad \wedge$
   $\texttt{u9} = \textit{any} \rightarrow \texttt{u6} = \textit{any} \vee a \quad \wedge$
   $\texttt{u6} = \textit{peer} \rightarrow \texttt{u9} = \textit{peer} \vee a \quad \wedge$
   $\texttt{u6} = \textit{rep} \rightarrow \texttt{u9} = \textit{rep} \vee a \quad \wedge$
   $a \leftrightarrow \texttt{u7} = \textit{any}$

Since `u9 Data` has no nested type arguments and there are no more type arguments on the same level, we are done.

4.4.4 Convertible Constraint

In order to understand the convertible constraint, we have to consider the equality operators $==$ and $! =$ which indicate if two references point to the same object. For that two references can refer to the same object, their main modifiers have to be in a subtype relationship. Thereby it does not matter which of the two ownership modifiers is the subtype and which is the supertype. It is enough if one can be converted — casted — to the other. This is represented by the convertible constraint.

Building a Convertible Constraint

For equality operators $==$ and $! =$, we take the *Type* to the left of the operator as input for *Type A* and the *Type* to the right for *Type B* for the following algorithm:

Building a Convertible Constraint

Input:	Two *Types* A and B
Output:	-
Side-effect:	A convertible constraint is added if applicable

1. If A is a *Type Variable*, take its upper bound until a *Non-variable Type* is reached. This then becomes A. Do the same for B.
2. We check if A's *Type Constraints* already contains B in the list of convertible constraints, or if B's *Type Constraints* already contains A in the list of convertible constraints. If either of them is the case, we are done.
3. Otherwise, we add *Non-variable Type* B to the list of convertible constraints of *Non-variable Type* A.

The algorithm is straightforward: In the first step, we substitute *Type Variable*s by their upper bounds for the same reason we do it in the other algorithms. Then, we check if a convertible constraint already exists for the two given *Type*s. Due to its symmetric property, we have ensure that it does not exist in either of the *Type Constraints*. If this is not the case, the constraint is added.

Listing 4.10: An example to explain the convertible constraint algorithms

```
 1  class GenericBase<T> {
 2  }
 3
 4  class GenericExtension<X, Y> extends GenericBase<X> {
 5  }
 6
 7  class ConvertibleConstraint {
 8          u1 GenericBase<u2 Data> obj1;
 9          u3 GenericExtension<u4 Data, u5 Object> obj2;
10
11          void main() {
12                  // initialization omitted
13
14                  if (obj1 == obj2) {
15                  }
16  }
```

We will show the application of the convertible constraint building algorithm with the help of the example given in Listing 4.10 where we present three classes. For the first two, `GenericBase` and `GenericExtension`, we have omitted the upper bounds of the *Type Variable*s, and thereby their *Variable*s. A possible convertible constraint is on line 14 where `u1 GenericBase<u2 Data>` is the input for A and `u3 GenericExtension<u4 Data, u5 Object>` for B. Let us apply the algorithm step-by-step:

1. Neither A nor B are *Type Variables*. Skip.
2. This convertible constraint does not exist in either of the *Type Constraints*'.
3. `u3 GenericExtension<u4 Data, u5 Object>` gets added as a convertible constraint to `u1 GenericBase<u2 Data>`'s *Type Constraints*.

Writing a Convertible Constraint

Writing the constraint is done by the algorithm below. The *Non-variable Type* to which the instance of *Type Constraints* belongs is A and the *Non-variable Type* retrieved from *Type Constraints* is B.

Writing a Convertible Constraint

Input:	Two *Non-variable Types* A and B
Output:	An arbitrary number of clauses in CNF representing the convertible constraint of A and B
Side-effect:	-

1. From A, take the main modifier. We call it u_0.
2. From B, take the main modifier. We call it u'_0.
3. Write the following constraints:
$$\begin{array}{ll} u'_0 = any \rightarrow u_0 = any \vee s & \wedge \\ u_0 = peer \rightarrow u'_0 = peer \vee s & \wedge \\ u_0 = rep \rightarrow u'_0 = rep \vee s & \wedge \\ u_0 = any \rightarrow u'_0 = any \vee \neg s & \wedge \\ u'_0 = peer \rightarrow u_0 = peer \vee \neg s & \wedge \\ u'_0 = rep \rightarrow u_0 = rep \vee \neg s & \end{array}$$
4. If A is not a subtype of B and B is not a subtype of A either, both based on Java?s subtyping rules, we are done.
5. Otherwise, if the *Class Information* of A and B refer to different classes, determine which of A and B is the subtype and which is the supertype with respect to Java's subtyping. Walk up the subtype's supertype hierarchy and substitute the type arguments until subtype and supertype refer to the same class. This then becomes A or B depending on what it was originally. It might be necessary to backtrack as there can be multiple supertypes due to interfaces.
6. For each type argument of A: If the type argument is a *Type Variable*, go to the next type argument. Otherwise, it is a *Non-variable Type*, take A's ownership modifier and call it u_i where i is the current nesting level. Take the corresponding modifier of B and call it u'_i. Then write the following clauses:
$$\begin{array}{ll} u_i = any \rightarrow u'_i = any \vee a & \wedge \\ u'_i = peer \rightarrow u_i = peer \vee a & \wedge \\ u'_i = rep \rightarrow u_i = rep \vee a & \wedge \\ a \leftrightarrow u_0 = any \wedge ... \wedge u_{i-1} = any \wedge \neg s & \wedge \\ u'_i = any \rightarrow u_i = any \vee a' & \wedge \\ u_i = peer \rightarrow u'_i = peer \vee a' & \wedge \\ u_i = rep \rightarrow u'_i = rep \vee a' & \wedge \\ a' \leftrightarrow u'_0 = any \wedge ... \wedge u'_{i-1} = any \wedge s & \end{array}$$
This is done recursively for nested type arguments, i.e. continue with nested type arguments before moving to the next type argument on the same nesting level. Thereby, A is replaced by the current type argument.

The algorithm is similar to the subtype constraint writing algorithm. Each time the algorithm is executed, a new literal s is introduced. The literal indicates which subtype relationship between the main modifiers of the two *Non-variable Types* is enforced: s being $false$ means that B's main modifier has to be a subtype of A's main modifier — s being *true* means that A's main modifier has to be a subtype of B's main modifier. That way it is up to the solver to assign s a boolean value and satisfy the remaining clauses. What remains to be fulfilled after s has been assigned a boolean value are the exact same clauses that would have been created by the subtype constraint for the main modifiers.

For the convertible constraint, we detach the subtype relationship of ownership modifiers from the subtype relationship of classes in Java. Reason being that one *Type* can be more specific on the object's class whereas the other *Type* is more specific what the universe is concerned. For example `peer Data` and `any ExtendedData` can refer to the same object although there is no subtype relationship between them in Generic Universe Types.

Step 4 terminates the algorithm if there is no subtype relationship between the two *Type*s in Java. Such a situation can arise if at least one of the *Type*s describes an interface. Due to multiple inheritance for interfaces in Java, the dynamic type of an object can be a subtype of two *Type*s that are in no subtype relationship. In this case, there is no constraints to be created for the type arguments.

If we have a subtype relationship between the two *Type*s but they are not equal what the class is concerned, we take the subtype and traverse its supertype hierarchy until the two *Type*s refer to the same class.

In step 6, we continue to enforce the subtype relationship between the ownership modifiers for each type argument. For each type argument being a *Non-variable Type*, two new new literals are introduced: a indicates if limited covariance were allowed for A's main modifier being the supertype — a' indicates it for B's main modifier being the supertype. Note that the literal that does not belong to the supertype is always $false$.

We illustrate the algorithm by continuing the example from Listing 4.10. From there we have `u1 GenericBase<u2 Data>` as input for A and `u3 GenericExtension<u4 Data, u5 Object>` for B:

1. $u_0 =$ `u1`

2. $u'_0 =$ `u3`

3. The following constraints are written:

   $$\begin{array}{ll}
   \texttt{u3} = \mathit{any} \rightarrow \texttt{u1} = \mathit{any} \vee s & \wedge \\
   \texttt{u1} = \mathit{peer} \rightarrow \texttt{u3} = \mathit{peer} \vee s & \wedge \\
   \texttt{u1} = \mathit{rep} \rightarrow \texttt{u3} = \mathit{rep} \vee s & \wedge \\
   \texttt{u1} = \mathit{any} \rightarrow \texttt{u3} = \mathit{any} \vee \neg s & \wedge \\
   \texttt{u3} = \mathit{peer} \rightarrow \texttt{u1} = \mathit{peer} \vee \neg s & \wedge \\
   \texttt{u3} = \mathit{rep} \rightarrow \texttt{u1} = \mathit{rep} \vee \neg s &
   \end{array}$$

4. B, `GenericExtension`, is a subtype of A, `GenericBase`. Skip.

5. From the previous step, we know that B is the subtype. We therefore walk up in the supertype hierarchy to its direct parent, `GenericBase<X>`, and substitute the type arguments. The resulting type is `u3 GenericBase<u4 Data>`. We assign it to B.

6. We take A's first type argument `u2 Data`. $u_1 =$ `u2` and correspondingly $u'_1 =$ `u4`. The following clauses are written:

   $$\begin{array}{ll}
   \texttt{u3} = \mathit{any} \rightarrow \texttt{u5} = \mathit{any} \vee a & \wedge \\
   \texttt{u5} = \mathit{peer} \rightarrow \texttt{u3} = \mathit{peer} \vee a & \wedge \\
   \texttt{u5} = \mathit{rep} \rightarrow \texttt{u3} = \mathit{rep} \vee a & \wedge \\
   a \leftrightarrow \texttt{u2} = \mathit{any} \wedge \neg s & \wedge \\
   \texttt{u5} = \mathit{any} \rightarrow \texttt{u3} = \mathit{any} \vee a' & \wedge \\
   \texttt{u3} = \mathit{peer} \rightarrow \texttt{u5} = \mathit{peer} \vee a' & \wedge \\
   \texttt{u3} = \mathit{rep} \rightarrow \texttt{u5} = \mathit{rep} \vee a' & \wedge \\
   a' \leftrightarrow \texttt{u4} = \mathit{any} \wedge s &
   \end{array}$$

Since `u2 Data` has no nested type arguments and there are no more type arguments on the same level, we are done. Note that no constraint involves `u5`, the ownership modifier of B's second type argument `u5 Object`. This makes sense as `u5` can be any of the three modifiers and B would always be convertible to A.

4.4.5 Cast Constraint

Although casts compromise the static type safety and therefore have to be avoided wherever possible, there are cases in Java where casts are almost inevitable, i.e. avoiding them implied unreasonably additional effort. We are minimizing the impact of casts on the static type safety of Generic Universe Types by defining a special kind of constraint: the cast constraint.

The goal is to constrain the *Variable*s of casts such that they have no influence on the ownership modifiers, i.e. although the class of a type changes, the ownership modifier remains the same. This is not always possible, as we will show.

Building a Cast Constraint

The following algorithm builds a cast constraint for two *Type*s A and B where A is cast to B, i.e. $(B)\ a$, a of *Type* A:

Building a Cast Constraint

Input:	Two *Type*s A and B
Output:	-
Side-effect:	A cast constraint is added if applicable

1. If B is a *Type Variable*, build a convertible constraint using the convertible constraint building algorithm with input A and B. We are done.
2. If A is a *Type Variable*, take its upper bound until a *Non-variable Type* is reached. This then becomes A.
3. Check if A's *Type Constraints* already contains B in the list of cast constraints. If this is the case, we are done.
4. Otherwise, add *Non-variable Type B* to the list of cast constraints of *Non-variable Type A*.

If B is a *Type Variable*, we do not know which *Type* we are casting to. All we know is that it is some subtype of the upper bound. By building a convertible constraint, we ensure that A and B have a common subset of types for which the cast succeeds. If we wanted to build stronger constraints, we would have to look at the type arguments and perform data flow analysis which required a lot of effort for a rare case.

Listing 4.11: An example to explain the cast constraint algorithms

```
class GenericBase<T> {
}

class GenericExtension<X, Y> extends GenericBase<X> {
}

class CastConstraint {
        u1 GenericBase<u2 Data> base;
        u3 GenericExtension<u4 Data, u5 Object> extension;

11      void main() {
12              // initialization omitted
13
14              extension = (u6 GenericExtension<u7 Data, u8 Object>) base;
15      }
16  }
```

In Listing 4.11 reuse the classes from the convertible constraint example. We apply the just presented algorithm to the cast on line 14. The *Type* to be cast, `u1 GenericBase<u2 Data>`, serves as input for A and the target *Type*, `u6 GenericExtension<u7 Data, u8 Object>`, for B.

1. B is a *Non-variable Types*. Skip.
2. A is a *Non-variable Types*. Skip.
3. Such a cast constraint does not yet exist.
4. We add `u6 GenericExtension<u7 Data, u8 Object>` to the list of cast constraints in `u1 GenericBase<u2 Data>`'s *Type Constraints*.

Writing a Cast Constraint

The following algorithm writes a cast constraint. The *Non-variable Type* the instance of *Type Constraints* belongs to is A and the *Non-variable Type* retrieved from *Type Constraints* is B.

Writing a Cast Constraint

Input:	Two *Non-variable Types* A and B
Output:	An arbitrary number of clauses in CNF representing the cast constraint of A being cast to B
Side-effect:	-

1. If the *Class Information* of A and B refer to different classes, determine which of A and B is the subtype and which is the supertype with respect to Java's subtyping. Walk up the subtype's supertype hierarchy and substitute the type arguments until subtype and supertype refer to the same class. This then becomes A or B depending on what it was originally. It might be necessary to backtrack as there can be multiple supertypes due to interfaces.
2. For each *Variable* u of *Type* A, take their counterpart u' of *Type* B. Write the following constraint:

$$\begin{aligned} & u = peer \leftrightarrow u' = peer && \wedge \\ & u = rep \leftrightarrow u' = rep && \wedge \\ & u = any \leftrightarrow u' = any \end{aligned}$$

As in the subtype constraint, we first make sure that both *Types* refer to the same class. We then create clauses that equate the corresponding *Variables* of the two *Types*. Alternatively, instead of forcing these pairs of *Variables* to be equal, we could also use the same *Variable* in both *Types*.

The idea is to only change the class and to not touch the ownership modifiers. That way, the cast has no impact on ownership, i.e. it is statically safe what the *owner-as-modifier* property is concerned. Yet, this is not true for all casts: If B contains type arguments that do not directly relate to one of A's type arguments, it will not be constrained by the cast constraint. Again, we would need data flow analysis to cope with such casts. So far, we expect that there will be other constraints like a subtype constraint built for the assignment that constrains these *Variables*.

We will illustrate the algorithm by going back to the example before. There we built a cast constraint with `u1 GenericBase<u2 Data>` as input for A, and `u6 GenericExtension<u7 Data, u8 Object>` as input for B. We now use the same input to write the constraints:

1. We take the base class of B which is `GenericBase<X>` and then substitute the type variable by its type argument `u7 Data`. This results in B being `u6 GenericBase<u7 Data>`.

2. A has two *Variable*s: `u1` and `u2`. Their counterparts in B are `u6` and `u7`. We therefore create clauses for these two:

   $\texttt{u1}= \textit{peer} \leftrightarrow \texttt{u6}= \textit{peer} \quad \wedge$
   $\texttt{u1}= \textit{rep} \leftrightarrow \texttt{u6}= \textit{rep} \quad \wedge$
   $\texttt{u1}= \textit{any} \leftrightarrow \texttt{u6}= \textit{any} \quad \wedge$
   $\texttt{u3}= \textit{peer} \leftrightarrow \texttt{u7}= \textit{peer} \quad \wedge$
   $\texttt{u3}= \textit{rep} \leftrightarrow \texttt{u7}= \textit{rep} \quad \wedge$
   $\texttt{u3}= \textit{any} \leftrightarrow \texttt{u7}= \textit{any}$

The example shows that no clause involving *Variable* `u8` is created. By considering only the cast constraint, it can adopt any of the three ownership modifiers. But this is not the only constraint dealing with the cast's target *Type*: the assignment to field `extension` creates a subtype constraint which includes clauses for *Variable* `u8` and ties it to `u5`. Depending on the later use of the second type argument, the solver finds an appropriate ownership modifier for `u5`. This then also has an impact on `u8`.

Universe Casts

By handling casts as we have presented them, we minimize the influence of casts on ownership and therefore are able to give strong static guarantees even in the presence of casts. We are convinced that programs can be annotated without casts — some might need refactoring or restructuring. This forces to reconsider the class and ownership structure and results in cleaner code.

In [21], Matthias Niklaus presents alternative ways to handle casts for the Universe Type System where casts were inevitable for bigger programs. He uses additional ownership modifiers to represent the `any` (`readonly`) modifier. These additional states record if the `any` modifier has just been assigned a `peer` or `rep` reference and therefore can safely be cast to `peer` or `rep`. To that end, he introduces several type systems that offer different levels of static safety. In return for the loss of static safety, more casts are allowed which increases the number of programs that can possibly be annotated. He also introduces *universe casts* — casts that do not change the class but only change the ownership modifier, e.g. from `any` to `peer` to allow calling a non-pure method. He presents in detail where universe casts are potentially useful.

Dealing with casts is always a trade-off between static type safety and flexibility. We have decided to reduce flexibility for the sake of static correctness and static type safety. We believe that developers who use our tools think in a similar way and that they are not satisfied with Java's static type safety. However, these alternative ways to annotate casts are certainly ideas to consider for the future.

4.5 Implementation

Due to time constraints we were not able to implement what we have presented in this chapter. In this section, we will give you an overview of the architecture of the existing tool, and explain how to extend it.

4.5.1 Architecture of the Existing Tool

The existing static universe inference tool was developed for the Universe Type System by Matthias Niklaus and integrated into Eclipse [5] by Andreas Fürer. It is written in Java. We will give an overview of the architecture. A detailed description can be found in [21] and [12]. Basically, the tool can be divided into three parts: the front-end, the UTI Interface, and the back-end.

The *UTI Interface* — UTI stands for Universe Type Inference — is a series of Java interfaces that decouple the front-end from the back-end. It allows to combine arbitrary front-ends with any back-end. Thereby the front-end makes use of the functionality offered by the interface whereas the back-end implements the interface. The most important interfaces are:

- `UtiController` lets the front-end control the back-end. For example, the front-end can tell the back-end to solve the current constraints and then have it return the found solution.
- `UtiConstraintBuilder` uses the builder pattern [13] to build the back-end's internal data model by telling it the structure of the program to be inferred, e.g. the methods it contains and how they are called.
- `UtiVariable` is the representative of an ownership modifier in the UTI interface. After a solution is found, it can be asked for its annotation.
- `UtiSolutionDescription` contains the annotation to the program. It can be queried to get all ownership modifiers.

The front-end — referred to as *client* in [21] — is responsible for handling the user interaction and controlling the back-end through the UTI interface. This part was changed by Andreas Fürer who integrated it smoothly into Eclipse and thereby made the tool a lot easier to use. The following components are key to the front-end:

- The JML compiler [18] parses the input and builds the abstract syntax tree which is used to extract the program structure from.
- Class `UniverseJmlVisitor` is a visitor for the abstract syntax tree which it traverses. While traversing the abstract syntax tree, it makes use of the `UtiConstraintBuilder` to pass the relevant parts to the back-end.
- Class `JmlToUtiMapper` contains several maps that maintain a mapping from elements of the abstract syntax tree to UTI objects. Every time an element of the abstract syntax tree is visited, a lookup is done in this mapping to reuse existing UTI objects.
- Class `UTAnnotationView` presents the found annotation to the developer in an Eclipse workbench view.

The back-end — called *inferrer* in [21] — implements the Uti interface, maintains an internal data model to store the program structure, creates constraints and solves them. The following components are most important:

- PBS [6] — a pseudo-boolean and MAX-SAT solver — solves the clauses being passed in CNF. Its proprietary file format is written by class `PbsCnfPbWriter`.
- Package `pbs_uti` contains implementations for all UTI interfaces. This includes various type systems to handle casts, and each type system has several ways to encode ownership modifiers on the bit level.
- Class `PbsCbNoCasts` implements `UtiConstraintBuilder` and builds the internal data model and constraints, without inserting universe casts. Its counterpart `PbsCbWithCasts` inserts universe casts at all places they can help to find a valid annotation.
- Class `PbsConstraints` maintains all constraints built for a variable.

4.5.2 Integrating Generic Universe Types

Integrating Generic Universe Types into the static universe inference tool requires changes to all three parts, i.e. the UTI interface, the front-end and the back-end. In the Universe Type System, there used to be a one-to-one relationship between types and ownership modifiers, i.e. each type had one ownership modifier. This is why the current implementation does not know of types but just of variables, represented by `UtiVariable`. For Generic Universe Types, the notion of types has to be integrated in the data model, e.g. by using the one we have presented in Figure 4.1.

Once there is a way to represent types — let us call it `UtiType` — many changes are focused around this class. In the UTI interface, most occurrences of `UtiVariable` have to be replaced by `UtiType`. The front-end then has to pass `UtiType`s instead of `UtiVariable`s to the back-end, and the back-end has to store constraints for `UtiType`s, and no longer for `UtiVariable`s.

As a next step, the existing constraints have to be replaced by the ones we have presented in Section 4.4. Since constraints are implemented for each type system separately, we suggest to start with the *static universe type system* implemented in `PbsBrStaticUniverse`. It corresponds to the one we have used for the constraints in Section 4.4 which makes the implementation straightforward. There is no need change the encoding of the ownership modifiers as they remain the same. Apart from the constraints, additional weights have to be introduced, e.g. for upper bounds of type variables, we would like to prefer `any`.

Eventually, the user interface has to be adapted to cope with multiple ownership modifiers for one type. Thereto, a reasonable way to display types has to be found. The XML annotation schema that stores the found annotations has to be updated as well and then the annotation tool has to be changed to insert the annotations.

We have created a proposal for the XML annotation schema. Apart from allowing multiple ownership modifiers for one type, we have added definitions of type variables for classes and methods and the superclass and implemented interfaces as they can now contain ownership modifiers. Furthermore, method calls are included since we have to explicitly specify the type arguments for method type variables. The example presented in Listing 4.4 is based on this proposal.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

We have presented an extended version of Generic Universe Types that can cope with most of Java's commonly used features. To that end, we have added static members, arrays and exceptions and extended defaulting to handle them in unannotated code. We have implemented all this in the MultiJava compiler which makes it also availabe to the JML tools.

We invite you to give Generic Universe Types a shot. It sure is a good way to structure the heap hierarchically. It makes you think more about the program you are writing and gives you a better understanding of its structure.

Futhermore, we have looked into raw types and how they can be dealt with in Generic Universe Types. We have explained wildcards and their application to Generic Universe Types. For universe wildcards, we have come to the conclusion that they might be useful to resolve the issue of combinations of ownership modifiers that cannot be exactly represented by the three current ownership modifiers. We have not implemented any of these concepts.

As the extension to this Master's thesis, we have convered static universe type inference and how to apply it to Generic Universe Types. We have presented the core of the data model and extended the existing constraints to handle generic types. We have also given a brief description how to implement it.

5.2 Future Work

5.2.1 MultiJava and JML

Support for generic types in MultiJava — and therefore JML — still lacks stability. Adding test cases helps revealing bugs and increasing reliability. Implementing raw types and wildcards as we have presented should be straightforward and doable with little effort.

MultiJava has a bug when it comes to reading classfiles that inherit from a generic class or interface. This renders most of the Java collection API unusable. Fixing that would greatly increase usability.

Implementing the inference algorithm for method type variables and extending it to Generic Universe Types would be of great benefit for the developer as it reduces the unnecessary overhead.

Inner classes are a concept of Java that we have not discussed. We do not expect it to be diffucult to add to Generic Universe Types.

Due to a misunderstanding, we did not look into storing ownership modifiers for generic types in classfiles. This forces MultiJava and JML to read and compile the source code of each imported class everytime. Including ownerhsip modifiers in classfiles would speed up the compilation process.

5.2.2 Raw Types and Wildcards

We have presented an informal description of the application of raw types and wildcards to Generic Universe Types. Formalizing and implementing the concepts has not been done yet.

Another idea that looks promising is the readable modifier introduced in Section 3.6. The wildcard approach can be combined the ideas presented by Daniel Schregenberger in his Master's thesis [24].

5.2.3 Static Universe Type Inference

Based on the presented data model and constraints, implementing static universe type inference should be a straightforward task. Yet, it would take a lot of time as there are changes and improvements necessary all over the code and cover several tools.

This area offers room for further research as to where to put universe casts and other approaches as to how to deal with casts in general. Furthermore, we have not covered the various features of Java discussed in Sections 2 and 3.

5.3 Acknowledgements

I would like to thank my supervisor Werner Dietl for his support throughout this thesis. His comments and insights have been extremely helpful and contributed big time. I would also like to say thanks to Prof. Peter Müller for giving me the opportunity to do my thesis in his group. It has been a great experience.

My biggest thank goes to my parents Heidi and Jean for their incredible encouragement during my entire life.

Bibliography

[1] An Introduction to Design by Contract by Eiffel Software. http://archive.eiffel.com/doc/manuals/technology/contract/.

[2] Javadoc of the Java 5 API. http://java.sun.com/j2se/1.5.0/docs/api/.

[3] Website of Eiffel Software, creators of the Eiffel programming language. http://www.eiffel.com.

[4] Website of the ANTLR parser generator. http://www.antlr.org.

[5] Website of the Eclipse development environment. http://www.eclipse.org.

[6] Fadi Aloul. PBS v2.1: Incremental Pseudo-Boolean Backtrack Search SAT Solver and Optimizer. http://www.eecs.umich.edu/~faloul/Tools/pbs/, 2003.

[7] Marco Bär. Practical Runtime Universe Type Inference. Master's thesis, ETH Zurich, 2006.

[8] Curtis Clifton, Todd Millstein, Gary T. Leavens, and Craig Chambers. MultiJava: Design rationale, compiler implementation, and applications. *ACM Transactions on Programming Languages and Systems*, 28(3), May 2006.

[9] W. Dietl, S. Drossopoulou, and P. Müller. Generic Universe Types. In E. Ernst, editor, *European Conference on Object-Oriented Programming (ECOOP)*, Lecture Notes in Computer Science. Springer-Verlag, 2007. To appear.

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

[11] W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology (JOT)*, 4(8):5–32, October 2005.

[12] Andreas Fürer. Combining Runtime and Static Universe Type Inference. Master's thesis, ETH Zurich, 2007.

[13] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Abstraction and reuse of object-oriented design. *Lecture Notes in Computer Science*, 707:406–431, 1993.

[14] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java Series)*. Addison-Wesley Professional, July 2005.

[15] Thomas Hächler. Statische Felder im Universe Type System (in German). ETH Zürich, 2004.

[16] Nathalie Kellenberger. Static Universe Type Inference. Master's thesis, ETH Zurich, 2005.

[17] Martin Klebermaß. An Isabelle Formalization of the Universe Type System. Master's thesis, ETH Zurich, 2007.

[18] G. Leavens and Y. Cheon. Design by Contract with JML. http://www.jmlspecs.org.

[19] Frank Lyner. Runtime Universe Type Inference. Master's thesis, ETH Zurich, 2005.

[20] Ted Neward. Java statics - When static is not static. http://www.javageeks.com/Papers/JavaStatics/JavaStatics.pdf, 2001.

[21] Matthias Niklaus. Static Universe Type Inference using a SAT-solver. Master's thesis, ETH Zurich, 2006.

[22] Mathias Ottiger. Runtime Support for Generics and Transfer in Universe Types. Master's thesis, ETH Zurich, 2007.

[23] Daniel Schregenberger. Runtime checks for the Universe Type System. ETH Zürich, 2004.

[24] Daniel Schregenberger. Universe Type System for Scala. Master's thesis, ETH Zurich, 2007.