MultiJava, JML, and Generics

Ovidio José Mallo

Semester Project Report

Software Component Technology Group
Department of Computer Science
ETH Zurich

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

SS 2006

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

The goal of this semester project is to extend the already existing support for generics in the MultiJava compiler by implementing special features of Java generics on top of it, namely *wildcards* and *raw types*, while also making the necessary modifications to JML. To that end, we will give a brief overview of Java generics and describe the main design decisions behind its specification before presenting a thorough discussion of wildcards and raw types along with several interesting examples which should provide the necessary insight into some of the subtleties of the parametric polymorphism as realized in the Java 5 language. This will give us the necessary theoretical and practical ground before proceeding to the discussion of the actual implementation.

Contents

Chapter 1

Introduction

1.1 The evolution of the Java language

Java is an object-oriented programming language developed at Sun Microsystems since the early 1990s. Despite several unsuccessful attempts at formalizing the Java technology through the ISO/IEC JTC1 standards body and later the ECMA standards consortium, Java still remains a proprietary *de facto* standard whose evolution is controlled by the Java Community Process[1] (JCP).

Extensions to the Java platform are driven by so-called Java Specification Requests (JSRs) which embody proposals for the development of new specifications. Examples of such JSRs include the Java language specification itself but also the individual additions to the Java programming language introduced in J2SE 5.0 which we will mainly focus on in this writing. Adding new features to the Java programming language typically implies different kinds of modifications to be applied to the Java platform:

- Modifications to the *Java Language Specification* which describe the actual language constructs along with their syntax and semantics.
- Modifications to the *Java Virtual Machine Specification* which describe the JVM support required to implement the desired language feature. Some language constructs, such as *generics*, require changes to the JVM specification while others are mere compile-time constructs which have no representation in a class file. An example of the latter kind is the *enhanced for loop* introduced in the latest version of the Java language.
- Modifications to the *Java platform APIs*. *Annotations* are an example of a language construct which requires the APIs to be adapted in order to e.g. allow querying the annotations at runtime via reflection.

In this writing, the new features of the Java 5 language will be discussed. Thereby, we will mainly concentrate on the changes to the Java Language Specification and also cover some aspects of the JVM Specification. Modifications to the platform's APIs resulting from the new language constructs will not be treated.

1.2 MultiJava and JML

MultiJava, as described in [7], is a conservative extension of the Java programming language that adds symmetric multiple dispatch and open classes. Multiple dispatch provides a means of dynamically deciding at runtime which of a set of overloaded methods to invoke based on the runtime types of all method arguments. This contrasts with the way Java treats method

[1] see http://www.jcp.org/

invocations where dynamic dispatching is only performed on the receiver of the method but not on its arguments. Multiple dispatch provides a solution to the binary method problem and the implementation of typical programming paradigms such as event handlers greatly benefits from such a feature. Open classes, on the other hand, allow the modular addition of new methods to an existing class without modifying the class' code. This solves the extensibility problem of object-oriented programming languages. Despite those significant extensions, MultiJava preserves Java's static modular typechecking properties.

JML (Java Modeling Language) is a specification language for Java. In its most basic use, JML serves as a design-by-contract language for Java, as described in [11], but it also supports full abstract modeling which allows to specify the behavior of Java modules without exposing any implementation details.

The MultiJava project[2] provides a compiler for the MultiJava programming language which translates MultiJava programs to regular Java bytecode which can be executed on any standard Java Virtual Machine. The compiler targets version 1.4 of the Java programming language and also offers partial support for Java generics, as described in [5]. The goal of this semester thesis is to add full support for wildcards and raw types to the already existing generics implementation in the MultiJava compiler and to make the generics support also accessible to JML tools.

[2] see http://sourceforge.net/projects/multijava

Chapter 2

Generics

2.1 Introduction

JDK 1.2 introduced the `Collection`s type hierarchy into the Java platform APIs which provides a common abstraction for a large set of data structures. By that time, in order for a collection to hold arbitrary data, the elements it contains had to be declared as being of type `Object`, the type sitting at the top of Java's type hierarchy. This contrasts the fact that whenever a collection is actually used in some client code, the programmer typically knows that *all* elements it contains are of some type more specific than `Object`. However, earlier versions of the Java language did not offer any support for declaring that e.g. a `List` does not contain arbitrary data, but only elements of type `String` while some other list is supposed to only contain `Integer`s. Instead, the elements extracted from any collection were always of type `Object` and therefore had to eventually be cast to the appropriate type before further processing. This not only introduced additional clutter into the code but also, and more importantly, gave rise to potential exceptions at runtime as down-casts are not statically checkable.

This situation has changed recently with the introduction of JDK 5.0 which adds parametric polymorphism – also known as *generics* – to the Java language. Generics support the declaration of types dependent on a set of type parameters which are provided whenever a variable of that type is declared. This allows to e.g. declare a list which is supposed to only contain `String`s – supported by the simple syntax `List<String>` – while at some other point we operate on a list of `Number`s (or subtypes of it). This support for explicitly specifying the programmer's intention of having a list of a specific type allows the compiler to statically[1] enforce this program property.

2.1.1 Terminology

In this writing, we try to follow the terminology for generics from [9].

Type variables are unqualified identifiers that are introduced by *generic class declarations*, *generic interface declarations*, *generic method declarations*, and *generic constructor declarations*. A *type parameter* is a type variable with an optional set of declared *upper bounds*.

A generic type declaration defines a set of *parameterized types*, which consist of the class or interface name followed by an actual *type argument* list. Every such parameterized type is termed an *invocation* of its corresponding generic type declaration.

A *raw type* is the name of a generic type declaration used without any accompanying actual type arguments.

[1] Note that Java's static typechecking system may actually miss some unsafe operations which lead to exceptions at runtime but *only* in the presence of so-called unchecked warnings which may be issued under certain circumstances. This is discussed in 2.4.

2.2 Generics in the Java language

Many object-oriented programming languages nowadays offer support for some kind of parametric polymorphism. However, even though the basic idea is always the same, concrete implementations may differ considerably. As an example, the C++ template mechanism resembles a kind of high-level macro in which a new class is created at compile-time for *every* concrete parameterization of a class template. It is then up to the C++ linker to share the code for identical class instantiations. This approach offers a great flexibility and typically leads to better efficiency, but suffers from an obvious problem of code bloat.

For typesafe languages supporting generics, such as Java and C#, the design decision of whether to retain the generic type information at runtime is probably one of the most crucial ones as it greatly influences the backward compatibility of the new language but also the flexibility offered by the generic type system.

As described in [10], the approach taken for adding generics to version 2.0 of the C# programming language was to provide full runtime support for the generic type information. This had to be done at the cost of not being backward compatible to old non-generic code. By contrast, in Java the main goal was to be able to run new generic code on an unmodified JVM. Therefore, no runtime support for generic types is provided[2]. The actual mapping from types of the generic type system to types supported by the JVM at runtime is called *type erasure* and is defined in [9] as follows:

- The erasure of a parameterized type $G<T_1, \ldots, T_n>$ 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.
- The erasure of every other type is the type itself.

We see that erasure simply throws away all type arguments to parameterized types while type variables are replaced (recursively) by the erasure of their leftmost bound. Additionally, all type parameter lists in type and method/constructor declarations are discarded to make them non-generic. The whole process results in type declarations which contain no generic type information at all which is what the JVM will work with at runtime. The fact of having type information which gets erased during compilation calls for a distinction between types fully supported at runtime – so-called *reifiable types* – and types which are merely supported at compile-time. The following types are *not* reifiable:

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

These correspond to the types which lose information during type erasure as defined above. However, note the special case of a parameterized type having only unbounded wildcards (discussed in detail in 2.3) as actual type arguments. This singular form of a parameterized type is considered to be reifiable as unbounded wildcards stand for no particular type meaning that no real information is lost even if they are discarded upon performing type erasure.

[2] However, the current generics implementation has been carefully designed to not preclude a possible future extension adding runtime support for generic types.

2.2.1 Unchecked warnings

The Java type system supports several constructs which are not statically checkable. These include *down-casts*, the *type comparison operator*, and *array store operations*. In order to ensure that the integrity of the JVM is never at risk, the type safety of these constructs is enforced by a runtime check which triggers an exception for every unsafe operation. However, due to type erasure, these runtime checks cannot be performed reliably if they involve non-reifiable types. The approach taken in the Java language to circumvent this shortcoming is to not allow (i.e. produce compiler errors) for some of these constructs while issuing so-called *unchecked warnings* for others for which it would be prohibitively restrictive to not support them at source code level.

Unchecked warnings are used to flag possibly unsafe operations which are permitted by the type system as a concession for greater flexibility even though type erasure makes it impossible to actually check them either statically or at runtime. As a simple working example, let us consider Listing 2.1.

Listing 2.1: Unchecked operations and heap pollution

```
1  import java.util.Vector;
2
3  public class Unchecked {
4      public static void unsafe(Object someVector) {
5          Vector<String> strings = (Vector<String>) someVector; // Unchecked warning
6          String string = strings.get(0); // ClassCastException
7      }
8
9      public static void main(String[] args) {
10         Vector<Integer> integers = new Vector<Integer>();
11         integers.add(new Integer(7));
12         unsafe(integers);
13     }
14 }
```

On line 5, we have a cast to a parameterized type which is permitted by the type system. However, an unchecked warning must be issued as the cast cannot be checked reliably at runtime since the JVM has no information about the concrete invocation of a parameterized type. Consequently, the only check which can be performed for that cast is to see whether the cast's source is of type `Vector`. In our concrete example, the type of the cast source is `Vector<Integer>` as can be seen by looking at the `main` method meaning that the cast check succeeds at runtime and the assignment on line 5 is performed. This in turn implies that before executing the statment on line 6, the variable `strings` of type `Vector<String>` refers to an object of type `Vector<Integer>` which is *not* a subtype of the former. This very special fact of having a variable of some parameterized type point to an object of a *different* invocation of the *same* generic type declaration is known as *heap pollution*. This is the price we have to pay for the lack of runtime support for generics. Heap pollution can, as illustrated on line 6, lead to a `ClassCastException` at runtime. As this is clearly undesirable, the Java Language Specification mandates to flag *every* operation which *may* lead to heap pollution with an unchecked warning. In other words, this means that, in the absence of unchecked warnings, accessing the object stored in a type variable will never lead to a `ClassCastException` as the one above.

It should be noted, however, that even in the presence of unchecked warnings the integrity and type safety of the JVM is never at risk. This is guaranteed by inserting implicit casts whenever an object stored in a type variable is accessed in some way. In particular, this means that every access to a variable whose type is a type variable and every method invocation whose return type is a type variable eventually requires an implicit cast to be inserted. Listing 2.2 shows different examples which illustrate these simple scenarios but also some eventually less apparent subtleties of the problem.

Listing 2.2: Implicit generics casts

```
1  import java.util.Iterator;
2
3  public class Casts<T extends Number & Cloneable & Iterable> {
4      public T foo(T t) {
5          Number number = t; // no cast
6          Cloneable cloneable = t; // cast to Cloneable
7          Iterable iterable = foo(t); // cast to Iterable
8
9          Iterator iterator = t.iterator(); // cast to Iterable
10
11         return null;
12     }
13 }
```

The type variable `T` in the example has three types declared in its bounds. Its erasure is the type `Number` which means that the object contained in the type variable is always of that type. Therefore, the assignment on line 5 requires no cast to be inserted. The other two types in the type variable's bounds, however, get erased during compilation meaning that it is not guaranteed that the actual type contained in the type variable really satisfies those bounds[3]. For that reason, prior to performing the assignments on the lines 6 and 7, an implicit cast to the types `Cloneable` and `Iterable`, respectively, needs to be inserted. This illustrates that the type to which a type variable needs to be cast not only depends on the type variable itself but also on the concrete context in which it is being used. In any case, that cast target is always one of the types declared in the type variable's bounds.

Finally, line 9 illustrates the invocation of a method on a type variable. In that case, the object contained in the type variable must eventually be cast to the type in which the method is declared as is indicated in our example.

It is thereby interesting to see that the implicit casts inserted by the compiler are exactly those which one would write explicitly if working with the erasure of the generic type declaration. In particular, this means that the use of generics in Java leads to no performance gain in that respect as it is not safe to omit those down-casts. This is different from e.g. generics in C# where these casts are superfluous due to the fact that generics are supported at runtime.

2.2.2 Example

Listing 2.3 illustrates the impact of type erasure on the usage of parameterized types and type variables.

Listing 2.3: Impact of type erasure on the type system

```
1  import java.util.Vector;
2
3  class Erasure<T extends String> {
4      public void erased(Object arg) {
5          Vector<String> vector1 = (Vector<String>) arg; // Unchecked warning
6
7          if (arg instanceof Vector<String>) { } // Error
8
9          Vector<String>[] array = new Vector<String>[10]; // Error
10         Object[] alias = array;
11         alias[0] = new Vector<Integer>(); // Array store should fail, but does not!
12         String element = array[0].get(0);
```

[3] at least not in the presence of raw types, as discussed in detail in 2.4

```
13
14          Vector<String> vector2 = new Vector<String>(); // OK
15          T t = new T(); // Error
16          Vector<T> vector3 = new Vector<T>(); // OK
17      }
18  }
```

Line 5 shows a cast to a parameterized type which is permitted by the type system even though an unchecked warning is issued. The reason for that is that this cast cannot be fully checked at runtime as the JVM has no information about concrete invocations of a generic type declaration. Therefore, the only possible runtime check for this cast is to verify whether the acutal object contained in the variable `arg` is of type `Vector`. As a general rule, for every cast to a non-reifiable type, only the conformance of the cast's source to the *erasure* of that type is actually checked.

On line 7, we see that the type comparison operator may not be applied to non-reifiable types as no information about generic type invocations or type variables is available at runtime. Note that this line really triggers a compiler error and not just an unchecked warning as the result of such a type comparison on a non-reifiable type would make only little sense.

The reason for not permitting the instantiation of the array on line 9 is probably a bit less apparent. However, let us consider what would happen on the subsequent lines if this was allowed. On line 10, we declare an alias for our array which we use on line 11 to store an object of type `Vector<Integer>` in the array. At this point, we would expect the mandatory array store check performed by the Java runtime to fail as the object inserted into the array is not compatible with the array's component type `Vector<String>`. However, the runtime check would succeed as, after erasure, the array as well as the object being stored in the array are both merely known to be of type `Vector`. Consequently, the statement on line 12 would be executed resulting in a `ClassCastException`. Hence, as every statement on the lines 10 to 12 by itself is clearly legal, the only way to avoid this kind of problems is to not permit the instantiation of arrays whose component type is not reifiable. This ensures that an array store check can always be performed reliably.

Finally, lines 14 and 15 illustrate a situation in which parameterized types and type variables are treated differently. As indicated on line 14, it is permitted to instantiate an object of a parameterized type. This poses no problem since a parameterized type is a concrete, known type which can be instantiated like any non-generic type. By contrast, a type variable is not bound to a fixed type but instead would depend on the concrete invocation of the current `this` object. However, as this information is not available after erasure, no object whose type is a type variable can be instantiated. Note that in turn having a type variable in the type arguments of a parameterized type is no impediment for instantiating an object of the latter type as, once more, the actual type arguments are erased and therefore have no influence on the object instantiation but are merely used for the subsequent static type checking. This is exemplified on line 16.

2.3 Wildcards

Up to now, whenever we wanted to use a generic type declaration, we always had to specify concrete type arguments for every declared type parameter. However, there are plenty of situations in which code written for some generic type is applicable to all, or at least a subset of, possible invocations of that type as it is independent of the concrete parameterization.

Wildcards, first described in [14], address this important use case of parameterized types by introducing a type safe abstraction over different invocations of a generic type. A wildcard is a special kind of type argument which stands for any and no particular type. Wildcards are supported by the suggestive syntax `List<?>` to e.g. express that we want to work on a list of some arbitrary type. Note that even though we would not know anything about the type of the elements stored in such a list, we would still be able to operate on the list itself by querying and also modifying the parts of the list's state which are independent of the concrete parameterization

of the list.

To support invocations in which the concrete parameterization is not totally irrelevant but where we need at least some partial knowledge about the type arguments, a wildcard can be equipped with an *upper bound* as in `List<? extends Number>`. While this still does not tell us what the actual type of the elements in the list is, we can at least be sure that it is a subtype of `Number` and use that information when working with such a list. Note that the idea of supporting an upper bound on wildcards is analogous to the notion of upper bounds on type parameters where the set of valid type arguments is restricted to a subset of the whole type hierarchy.

Additionally, wildcards can also be used in conjunction with a *lower bound*. In that case, a `List<? super Integer>` represents all possible invocations of a list whose type is a *supertype* of `Integer`. Note that this is very specific to wildcards and not supported on type parameters. Even though this feature is certainly more rarely used, let us consider a simple example taken from Java's platform APIs to see what flexibility lower bounded wildcards add to the type system by looking at the signature of the polymorphic `copy` method in the `Collections` class which reads

`public static` <M> `void` copy(List<? `super` M> dest, List<? `extends` M> src);

In this example, the use of a lower bounded wildcard permits to correctly express the semantic requirement that the element type of the `dest` list must be a supertype of the element type of the `src` list. Note that, even though more expressive, in this particular example one could afford the use of a lower bounded wildcard and instead declare the type of `dest` to simply be `List<M>` as `M` would be inferred to the type of the parameter's type argument, thereby allowing the same set of parameters to the method as if using the lower bounded wildcard. However, there are situations in which lower bounded wildcards are inevitable to express the semantic relationship between different parameters of a polymorphic method. An example for this is the `Collections.fill` method which replaces all elements in a given list by the specified object. Its signature

`public static` <M> `void` fill(List<? `super` M> list, M object);

expresses the fact that the element type of the given `list` must be a supertype of the type of the `object` to fill in. These semantics are solely captured by lower bounded wildcards.

2.3.1 Subtyping among parameterized types

Upon introducing generics into the Java language, the subtyping relationship among different types had to be extended to handle the presence of parameterized types. Historically, the addition of wildcards to the generic type system followed that of type parameters and type variables. For that reason, we try to mimic that evolution at this point by first discussing what the subtype relationship between concrete invocations (i.e. with no wildcards as type arguments) is and only then we will see what wildcards contribute to the flexibility of the type system.

Subtyping in the absence of wildcards

In short, *different* concrete invocations of a generic type declaration are *not* subtypes of each other. This means that all type arguments must be the *same* in both types. This is a very restrictive and counterintuitive yet necessary limitation. To see why, let us assume this restriction was not imposed by the type system and consider what would then happen in the example of Listing 2.4.

Listing 2.4: Generics and subtyping

```
1 import java.util.Vector;
2
3 public class Subtyping {
4     public void generics() {
5         Vector<String> strings = new Vector<String>();
6         Vector<Object> objects = strings; // Error
7
```

```
8           objects.add(new Object()); // storing an Object in a String vector!
9           String string = strings.get(0);
10      }
11
12      public void arrays() {
13          String[] strings = new String[1];
14          Object[] objects = strings; // OK
15
16          objects[0] = new Object(); // ArrayStoreException at runtime!
17      }
18  }
```

On line 6, we have the relevant assignment between parameterized types whose type arguments are not the same. We see that, if this assignment was permitted, we could use the alias of type `Vector<Object>` to insert a value of type `Object` into a vector whose element type is `String`, as illustrated on line 8. This is clearly wrong and would lead to a `ClassCastException` on the next line. Once more, the reason for this kind of problem is type erasure: as we do not have runtime information about the concrete invocation the variable `objects` is pointing at, we cannot check whether inserting an instance of type `Object` into it is valid or not. Note that this clearly contrasts the use of arrays which resemble parameterized types in that they also serve as containers for objects of some specified type (the array's component type). However, the key difference is that arrays in fact carry information about their actual component type at runtime. Therefore, covariant subtyping between arrays can be safely permitted (see line 14) as the array store on line 16 can be checked reliably at runtime, thereby avoiding spurious `ClassCastException`s.

Subtyping in the presence of wildcards

As was already mentioned, wildcards provide an abstraction over different invocations of a generic type declaration. This is fundamental for situations in which code operating on a parameterized type needs only partial knowledge about the actual parameterization of the type. As an example, assume we want to write a utility method which iterates over a list of listeners and notifies them of some event. Using wildcards, the signature of the method would look like

`void` notifyAll(List<? `extends` Listener> listeners, Event event);

Such a method could then be invoked with a list whose element type is an arbitrary subtype of `Listener` as actual parameter. As discussed above, this would not be the case if the type of `listeners` was the concrete invocation `List<Listener>`. In that case, only a variable of the exact same type could be passed as actual parameter even though the actual implementation would be able to operate on any type of listener.

It is thereby interesting to see that a method as the one above could also be expressed without using wildcards. Indeed, the polymorphic method

<M `extends` Listener> `void` notifyAll(List<M> listeners, Event event);

is semantically equivalent to the one above in that both accept the same set of actual type parameters. We see that the wildcard is simply replaced with a dummy type variable which has the same upper bound as the wildcard. However, other things expressible with wildcards cannot be simulated using type parameters, the most important one being the ability to declare *fields* which can hold objects of different invocations of some generic type declaration. Such a field could e.g. serve to store our list of listeners used above by giving it the type `List<? extends Listener>`. Note that something equivalent cannot be expressed without wildcards.

Type argument containment

At the beginning of this section, we have already mentioned that different *concrete* invocations of the same generic type declaration are not subtypes of each other. We have also seen that in the

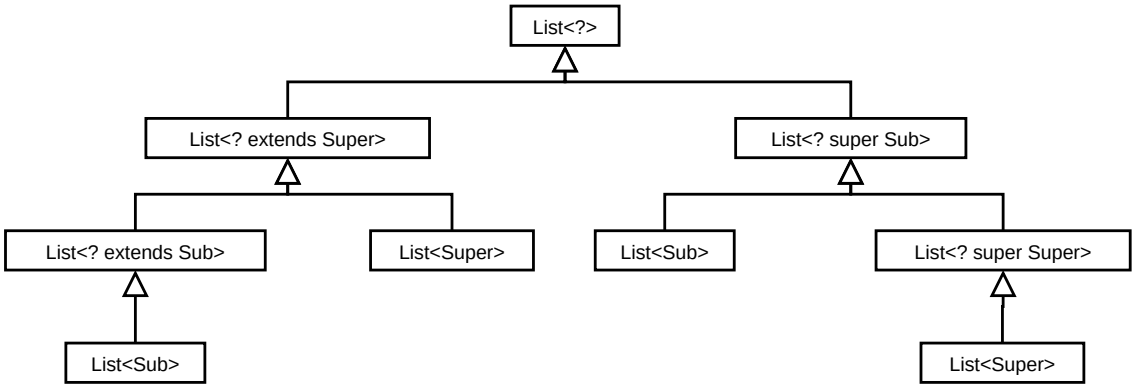

Figure 2.1: Subtyping among parameterized types; assuming Sub <: Super

presence of wildcards, this is different, but we have not yet discussed the exact rules for subtyping when using wildcards. This is what we will do at this point.

To that end, let us first define a special relationship between type arguments called *type containment*. As defined in [9], a type argument A_1 is said to *contain* another type argument A_2, written $A_2 <= A_1$, if the set of types denoted by A_2 is a subset of the set of types denoted by A_1. The type containment rules for concrete type arguments and wildcards read as follows:

1. ? extends T <= ? extends S, if T <: S
2. ? super S <= ? super T, if T <: S
3. T <= T
4. T <= ? extends T
5. T <= ? super T

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

By rule 1, we see that *upper* bounded wildcards exhibit *covariant* subtyping with respect to their bounds. Conversely, following rule 2, *lower* bounded wildcards give rise to *contravariant* subtyping. Rule 3 reflects the already discussed fact that different *concrete* invocations of a generic type declaration are not subtypes of each other as a non-wildcard type argument only contains itself. Rules 4 and 5 are used as a bridge between wildcards and concrete type arguments. One could e.g. use first rule 1 and then rule 4 to derive the following (note that the actual derivation goes from right to left):

Vector <= ? extends Vector <= ? extends Collection

Figure 2.1 graphically illustrates the subtype relationships between different invocations of the same generic type declaration starting at the parameterization with an unbounded wildcard and going down to concrete invocations where subtyping ends. Note that, even though depicted as such, the structure representing the subtype relationships is *not* a tree since concrete invocations are subtypes of different wildcard-parameterizations of a generic type. This immediately follows from the type containment rules 4 and 5 above.

2.3.2 Capture conversion

Wildcards by themselves are no real types as the set of types a wildcard covers also depends on the corresponding formal type parameter. In other words, an unbounded wildcard does not just stand

for an arbitrary type but rather for any type which respects the declared upper bounds of the wildcard's formal type parameter. Likewise, a bounded wildcard represents all the types which fulfill the constraints imposed by both the type parameter's upper bounds and the wildcard's upper or lower bound.

This dependence on a concrete type parameter is thereby the reason why wildcards can only be used as type arguments, since using them as types of variables or methods would be completely meaningless. However, even though we cannot directly specify wildcards as types in the code, whenever an expression is evaluated which accesses a member of a parameterized type, all eventual type variables contained in the member's type are substituted by the actual type arguments. If the type of the member is itself a type variable and the corresponding actual type argument is a wildcard, the type of the expression is also a wildcard. After such a generic type argument substitution, the wildcard would be completely detached from its formal type parameter meaning that type information would get lost. For that reason, we must make sure that this kind of generic substitution carries on all the type information contained not only in the wildcard but also in its formal type parameter. A very elegant solution to this problem is given by a process called *capture conversion* which was first described in [14] and later elaborated in [13].

The basic idea behind capture conversion is to ensure that *prior* to some member access on a parameterized type as the one described above, all *top-level* wildcard arguments of that type are replaced by a *newly created* special type variable which retains all the type information of the wildcard's formal type parameter as well as that of the wildcard bound itself, if any.

Capture conversion is a mapping from parameterized types to parameterized types in which all non-wildcard type arguments are left unchanged while wildcards are replaced by a special kind of type variable which is equipped not only with adequate upper bounds, as usual, but also with a lower bound. Such a type variable represents all the types which are subtypes of its upper bounds and, at the same time, supertypes of its lower bound.

Formalization of capture conversion

Following [9], let us assume that the parameterized type on which we want to apply capture conversion is $G<T_1, \ldots, T_n>$ and that its corresponding formal type parameters $A_1, \ldots, A_n$ have the declared upper bounds $U_1, \ldots, U_n$. Then, a new parameterized type $G<S_1, \ldots, S_n>$ results from capture conversion by transforming every type argument T_i to the new type argument S_i by one of the following rules ($U_i[A_k := S_k]$ denotes generic type substitution on U_i):

- If T_i is an unbounded wildcard, S_i becomes a *fresh* type variable whose upper bounds are $U_i[A_1 := S_1, \ldots, A_n := S_n]$ and whose lower bound is the null type.
- If T_i is a wildcard with the upper bound B_i , S_i becomes a *fresh* type variable whose upper bounds are B_i & $U_i[A_1 := S_1, \ldots, A_n := S_n]$ and whose lower bound is the null type.
- If T_i is a wildcard with the lower bound B_i , S_i becomes a *fresh* type variable whose upper bounds are $U_i[A_1 := S_1, \ldots, A_n := S_n]$ and whose lower bound is B_i.
- Otherwise, T_i is not a wildcard and we have $S_i = T_i$.

We see that an unbounded wildcard by itself carries no type information and, therefore, the type variable created for it simply has the same upper bounds as the wildcard's formal type parameter. A wildcard with an upper bound, by contrast, retains its full type information by including that bound in the upper bounds of the new type variable. A lower bounded wildcard contributes no type information to the upper bounds of the type variable being created, but its lower bound becomes the lower bound of the new type variable.

The lower bound thereby plays an important role for checking whether a value of some type can be assigned to a given type variable. More concretely, the Java Language Specification defines a type variable as being a direct supertype of its own lower bound. Consequently, a value of some type can only be assigned to a type variable if that type is the type variable itself or if it can be assigned to the type variable's lower bound.

Above, we see that unbounded and upper bounded wildcards both result in a type variable whose lower bound is the null type meaning that only `null` or a value of the same type variable can be assigned to them. This is intuitively clear since we do not know the concrete type captured by the wildcard and, therefore, we cannot be sure that the type we want to assign is really a subtype of that unknown type. However, type variables resulting from a lower bounded wildcard can be assigned arbitrary subtypes of its lower bound. This is always safe since, whatever the concrete type captured by the wildcard is, it is always a supertype of its lower bound.

When capture conversion needs to be applied

In order to avoid that the type of any expression in the code ever evaluates to an uncaptured wildcard, capture conversion must be applied to every parameterized type *before* one of its members can actually be accessed. To ensure this, capture conversion must be performed on the types of the following kinds of expressions: simple name expressions, field access expressions, method invocations, assignment expressions, cast expressions, array access expressions, and conditional expressions. Listing 2.5 shows some simple examples of applying capture conversion.

Listing 2.5: Applying capture conversion

```
1  public class Capture<T extends Number> {
2      T t;
3
4      Vector<T> vector;
5
6      public void apply(Capture<?> capture) {
7          Number number = capture.t;
8
9          capture = new Capture<Integer>();
10
11         Vector<?> vec = new Vector<String>();
12         capture.vector = vec; // Error
13     }
14 }
```

On line 7, we see that, without capture conversion, the type of the expression `capture.t` would evaluate to an unbounded wildcard. However, if capture conversion is applied correctly, *prior* to accessing the field `t`, we would convert the wildcard in the type `Capture<?>` of the simple name expression `capture` into a fresh type variable whose upper bound is the declared upper bound of the formal type parameter, i.e. `Number`. This type variable would then, after generic type substitution, become the type of the field being accessed, thus making the assignment valid. Here, we see how the type variable created during capture conversion carries the type information of the wildcard's formal type parameter in its upper bounds.

On line 9, we have an assignment between parameterized types. Before an assignment is actually evaluated, the type of the right hand side of the assignment must have been computed but not that of the left hand side as it is not needed for simply writing the variable. Similarly, prior to the actual assignment, capture conversion is only applied to the right hand side. This is important, as it ensures that such an assignment succeeds since `Capture<?>` is a supertype of `Capture<Integer>`. However, if capture conversion were also applied to the left hand side, the wildcard would be replaced by a type variable and, as was discussed in 2.3.1, a non-wildcard invocation is not a supertype of any other type.

Finally, on line 12, we see an assignment which is not permitted. Following the rules above, capture conversion is applied to the two simple name expressions `capture` and `vec` before actually performing the assignment. If we give the synthetic type variables created during capture conversion explicit names, this would result in `capture` having the type `Capture<T_01>` and `vec` the type `Vector<T_02>`. Please note that capture conversion is really applied individually on every

expression, thereby resulting in two *different* type variables – here `T_01` and `T_02` – being created. If we now perform generic type substitution on the field access `capture.vector`, we see that the assignment tries to copy a value of type `Vector<T_02>` into a variable of type `Vector<T_01>`. Since these are two different, concrete invocations of a generic type declaration, they are not compatible, as explained in 2.3.1. Here we see the importance of really creating a *fresh* type variable whenever capture conversion is applied.

Despite the application of capture conversion as described above, two restrictions on the use of wildcards must be imposed in order to definitely ensure that the type of an expression never evaluates to an uncaptured wildcard. These restrictions forbid to use wildcards as top-level type arguments in the type of a constructor call as well as in the supertypes of a type declaration.

The intuition behind capture conversion

Whenever a variable of some wildcard-parameterized type is declared in the code, the wildcard really stands for *any* and *no particular* type. However, as soon as the variable is read, we can be sure that, at some point in the code, the variable has been assigned a value of a *concrete* invocation of the generic type. After such an assignment, the type behind the wildcard is not anymore arbitrary, but it is some *fixed* yet unknown type.

Capture conversion provides a mechanism to differentiate between wildcards which are bound to a specific type and those which still stand for some arbitrary type. Bound wildcards are thereby transformed into special type variables as described above while unbound wildcards are left unchanged. As was already discussed, capture conversion only transforms *top-level* wildcards since these, and only these, are known to be bound to a specific type. To see why, let us consider the example in Listing 2.6.

Listing 2.6: Bound and unbound wildcards

```
1  public class A<T> {
2      T t1;
3      T t2;
4
5      public void foo() {
6          A<?> bound = new A<Integer>(); // ? is bound to Integer
7          bound.t1 = ”string”; // Error
8          bound.t2 = new Integer(7); // Error!
9
10         A<A<?>> unbound = new A<A<?>>(); // ? remains unbound
11         unbound.t1 = new A<String>(); // OK
12         unbound.t2 = new A<Integer>(); // OK
13     }
14 }
```

On line 6, we have an example of a top-level wildcard. We see that even though, in general, a variable of type `A<?>` can hold a value of *any* particular invocation of `A`, we just know that after the assignment, the wildcard represents a specific type, in our case `Integer`.

On the lines 7 and 8, we have two invalid assignments. The one on line 7 is intuitively wrong since we are trying to write a `String` into a variable which we know to be of type `Integer`. However, the assignment on line 8 is also forbidden since even though we know that, *in our particular example*, `bound.t2` is of type `Integer`, this cannot be guaranteed on a general basis since we merely know that the wildcard is bound to *some* type but not which concrete type it is.

Line 10 shows an example where the wildcard is *not* a top-level type argument. Already here, we see that after the assignment, the wildcard is still unbound. In fact, it is not possible to assign any other type to the variable since `A<A<?>>` is a *concrete* parameterized type and therefore, by the type containment rules in 2.3.1, incompatible with any different invocation of `A`. This immediately

shows why only top-level wildcard type arguments can be bound to a specific type which is what capture conversion reflects.

Finally, lines 11 and 12 show two different assignments to a variable of type `A<?>`. Since the wildcard is still unbound in both cases, both assignments succeed.

2.3.3 Wildcard capture

When invoking a polymorphic method, we can either explicitly specify the type arguments for the method type variables or, more commonly, we simply let the compiler infer them based on the actual parameters passed to the method. However, special care must be taken when some inferred type argument is a wildcard. In order to see how this can lead to problems, let us consider the example in Listing 2.7 which illustrates the two types of issues which can arise in conjunction with wildcards.

Listing 2.7: Type inference and wildcards

```
1  public class Inference {
2      public static <M> void reduce(Stack<List<M>> lists) {
3          List<M> top = lists.pop();
4          lists .peek().addAll(top);
5      }
6
7      public static <M> void append(List<M> list1, List<M> list2) {
8          list1 .addAll( list2 );
9      }
10
11     public static void main(String[] args) {
12         List<?> strings = Collections.singletonList ("string");
13         List<?> integers = Collections.singletonList (new Integer (7));
14
15         Stack<List<?>> lists = new Stack<List<?>>();
16         lists .push(strings );
17         lists .push(integers );
18         reduce( lists ); // Error
19
20         append(strings, integers ); // Error
21     }
22 }
```

On the lines 12 and 13, we create a list of strings and a list of integers, respectively. Both are stored in a variable whose static type is `List<?>`.

On line 15, we then create a stack whose elements are declared to be of type `List<?>`, meaning that we are allowed to safely push our two lists onto it as is done on the lines 16 and 17. On line 18, we have a polymorphic method call without explicit type arguments. Therefore, the compiler must infer the type of the type variable `M` declared in the method `reduce` in order to check whether the method invocation is valid or not. We see that in order for the method call to be valid, the compiler would have to infer the type variable `M` to be the passed wildcard. However, since the wildcard stands for some unknown type, this is not safe, as can be seen in our example. In fact, if the method call was permitted, the method `reduce` would be able to append the list of integers to the list of strings. The actual problem is that the method sees all elements stored in the stack as being of the *identical* type `List<M>` while the type `Stack<List<?>>` clearly permits to push lists having different element types onto the stack.

Finally, line 20 illustrates a similar situation where the two lists are passed separately. Likewise, instantiating the type variable `M` to a wildcard is again not permitted as it would lead to the same problems as those of the previous method call.

However, there are also situations in which it is perfectly safe to instantiate a type variable to a wildcard during type inference. As an example, let us consider the polymorphic method

<M> List<M> copy(List<M> list);

which returns a copy of a given list. For such a method, it is intuitively clear that passing a list of type `List<?>` is always safe, since even though we do not know the exact element type of the list, we can still be sure that it is *some* specific type and *any* such element type would make the invocation typesafe. Therefore, such a method call would be valid. The process of instantiating a type variable to a wildcard in methods similar to the one above is known as *wildcard capture*.

It is thereby interesting to see that the decision of whether a type variable can be instantiated to a wildcard during type inference or not becomes trivial in the presence of capture conversion. In fact, it is sufficient to only permit a type variable to be instantiated to a wildcard which has previously been transformed into a type variable during capture conversion. Intuitively, this means that a type variable may only capture wildcards which are known to be bound to some specific type.

Following this simple rule, we see that the invocation of the method `reduce` previously discussed in Listing 2.7 would indeed not be valid since the wildcard in `Stack<List<?>>` is *not* a top-level type argument and therefore, it would not get transformed during capture conversion.

The call to the `append` method is a bit different. In that case, two parameters are passed to the method, both of type `List<?>`. Since here the wildcards indeed *are* top-level type arguments, the process of capture conversion would transform each of them into an appropriate type variable. However, the two type variables are different and therefore not compatible to each other, meaning that the type inference for `M` would fail, thus making the method call invalid, as expected.

Passing a `List<?>` to our `copy` method above, by contrast, would be valid since the top-level wildcard gets transformed during capture conversion and, consequently, the type variable `M` can be instantiated to the wildcard.

2.4 Raw types

The Java language permits to use the name of a generic type declaration without specifying any accompanying actual type arguments, i.e. the erasure of a parameterized type is also considered to be a valid type. Such a type is termed a *raw type* and its interplay with a generic type system was first described in [3].

Supporting raw types is thereby a mere concession for being able to interoperate with legacy code (i.e. old non-generic code). Such a support is important since, otherwise, some generic client code in which e.g. the generic type `List<E>` is used could not take advantage of a non-generic library to operate on such a list as the library would address the list through its raw type `List`. Instead, one would have to wait until the old library is generified or, alternatively, write the new code without using generics. Even worse, two versions of the library would eventually have to be maintained if supporting generic as well as non-generic client code was desired. By contrast, if raw types are supported, it is e.g. possible to assign a parameterized type to its corresponding raw type. As we will see later, an assignment in the opposite direction is potentially unsafe but if we want to support legacy code, the type system must nevertheless permit it.

2.4.1 Raw type members

As a raw type has no type arguments, its supertypes and the types of its members are not well-defined by the usual generic type substitution used on parameterized types.

Instead, those types must be defined explicitly. Assuming that our raw type R belongs to the generic type declaration G, those types are defined as follows:

- The supertypes of R are the erasures of the supertypes declared in G.

- If a *non*-static member of R is declared in G, its type is the erasure of the member's declared type.
- The type of a *static* member of R is the same as its declared type.

In the definition, we see that the type of a non-static member is only erased if the member is really declared in the class corresponding to the raw type. If, by contrast, the member is defined in one of the supertypes of the raw type, we have to check whether that supertype itself is a raw type in order to decide whether to erase the member's type or not. An example of why this is really different is shown in Listing 2.8.

Listing 2.8: Raw type members

```
1  class Super {
2      public List<Integer> superList;
3  }
4
5  public class Sub<T> extends Super {
6      public static void main(String[] args) {
7          Sub raw = new Sub();
8          List<Integer> list = raw.superList; // access to non-raw-type member
9      }
10 }
```

On line 8, we see that the member `superList` is accessed on a variable of the raw type `Sub`. However, that member is not declared in `Sub` but rather in its supertype `Super` which is clearly *not* a raw type. Therefore, the type of the expression `raw.superList` is not erased and remains `List<Integer>`.

Figure 2.2 illustrates a few examples of how the member types of a raw type differ from those of its corresponding generic type. Thereby, it is interesting to note that, for non-static members, even types which do not depend on any class type variable get erased. This is e.g. the case for the variable `listString` whose type is `List<String>`. Static members, by contrast, preserve all their type information, including parameterized types and method type variables.

```
class G<T> extends Vector<T> {
    T t;

    List<T> listT;
    List<String> listString;

    <M> M foo(List<T> list);

    static List<String> staticList;
    static <M> void bar(M m);
}
```

```
class G extends Vector {
    Object t;

    List listT;
    List listString;

    Object foo(List list);

    static List<String> staticList;
    static <M> void bar(M m);
}
```

Figure 2.2: Class members of a generic type (left) and its corresponding raw type (right)

2.4.2 Type safety

As was already mentioned, raw types are merely supported to facilitate the interoperability of generic code with legacy code. However, except from this very special use case, using raw types should be avoided whenever possible as in the presence of raw types all the type safety guarantees otherwise provided by the generic type system become void.

In 2.2.1, we have already encountered a situation in which type safety cannot be totally guaranteed by the generic type system, namely in the presence of down-casts which involve non-reifiable types. Even though this shortcoming is inevitable due to the translation by type erasure employed in Java generics, we have seen that type safety can only be compromised at runtime if the compilation produced *unchecked warnings*. In order to ensure that this is still true in the presence of raw types, we have to identify all possibly unsafe operations introduced by raw types and flag them with an unchecked warning. To that end, we will first discuss an example which illustrates how raw types can actually undermine the type safety guarantees provided by the generic type system before giving the exact rules as of which operations require unchecked warnings to be issued. The example is given in Listing 2.9.

Listing 2.9: Unsafe operations with raw types

```
1  public class Unsafe<T> {
2      T t;
3
4      T getT() { return t; }
5
6      void setT(T t) { this.t = t; }
7
8      public static void main(String[] args) {
9          Unsafe<String> nonRaw = new Unsafe<String>();
10
11         Unsafe raw = nonRaw; // OK
12
13         raw.t = new Object(); // Unchecked warning
14         String s1 = nonRaw.t; // ClassCastException at runtime
15
16         raw.setT(new Object()); // Unchecked warning
17         String s2 = nonRaw.getT(); // ClassCastException at runtime
18
19         nonRaw = raw; // Unchecked warning
20         String s3 = nonRaw.t; // ClassCastException at runtime
21     }
22 }
```

On line 11, we have an assignment from a non-raw-type to a raw type. Such an assignment by itself is safe since the member types of any particular invocation of a generic type are always compatible to their own erasure. Hence, no unchecked warning is required. However, such an assignment nevertheless introduces a raw-type-alias for our parameterized type which, as a consequence, may lead to problems as is illustrated on the subsequent lines.

On line 13, that alias is used for writing a field whose *declared* type is the type variable `T`. However, as the field is accessed on a raw type, its type gets erased to `Object` meaning that the assignment is valid. This now clearly poses a problem because we know that the variable `raw` is actually an alias for an object of type `Unsafe<String>` which, in turn, expects its field `t` to contain an object of type `String`. As this is obviously not the case after our field assignment, the statement on line 14 would trigger a `ClassCastException` at runtime. The actual operation which leads to that exception is thereby the field assignment, but as its validity cannot be checked at runtime due to type erasure, the only thing we can do is issuing an unchecked warning in order to advice the user that he is performing a potentially unsafe operation which cannot be checked.

Line 16 illustrates a similar but nevertheless slightly different situation. This time, we are invoking a method on a raw type. Since the formal parameter type `T` gets erased to `Object`, the method call is valid. However, this is somewhat different from the field assignment above, since passing an instance of type `Object` to a parameter of type `T` does not automatically mean that this object is really stored in the raw type thereby leading to a `ClassCastException` when accessing

that object through a parameterized type. However, an unchecked warning must nevertheless be issued since the method body *may* actually store that object in the raw type, as is indeed done in the `setT` method, thus leading to the exception on line 17.

Finally, line 19 contains an assignment from a raw type to a parameterized type. Since, in general, the member types of some invocation of a generic type are more specific than their own erasure, such an assignment is potentially unsafe and must be flagged with an unchecked warning, since it may lead to a `ClassCastException` as the one on line 20.

After this more intuitive approach at explaining when and why an unchecked warning must be issued in the presence of raw types, we still provide the formal rules for the problem as they cover some aspects not discussed in the above example. These rules read as follows:

- An assignment from a raw type to a compatible parameterized type always triggers an unchecked warning.
- An invocation of a method or constructor of a raw type triggers an unchecked warning if any of the *declared* parameter types differs from the corresponding parameter type of the raw type.
- An assignment to a field of a raw type triggers an unchecked warning if its *declared* type differs from the corresponding type of the raw type.

In particular, this implies that *no* unchecked warning is generated when reading a field or when invoking a method in which only the declared return type differs from the corresponding result type of the raw type. In addition, the use of static members never triggers an unchecked warning as their declared types are not affected when accessed on a raw type.

Note also, that it is not always immediately apparent that some member is indeed being accessed on a raw type or that what is being assigned to a parameterized type in truth is a raw type. Listing 2.10 illustrates such situations.

Listing 2.10: Raw type members

```
1  class Super<T> {
2      public List<T> superList;
3  }
4
5  public class Sub extends Super {
6      public static void main(String[] args) {
7          Sub nonRaw = new Sub();
8          List<Integer> list = nonRaw.superList; // Unchecked warning
9
10         Super<String> sup = new Sub(); // Unchecked warning
11     }
12 }
```

In the example, we see that the class `Sub` extends the *raw type* `Super`. Therefore, the field access on line 8 is indeed an access to a member of a raw type, even though, in the code, we are accessing it on the non-raw-type `Sub`. Similarly, on line 10, we are in fact assigning the raw type `Super` to the parameterized type `Super<String>`, thereby leading to an unchecked warning to be issued.

2.4.3 Raw types in the Java platform APIs

As we have just seen, raw types bring us back many of the type safety issues we were faced to before the introduction of generics into the Java programming language. For that reason, their use should be avoided whenever possible. However, there are some situations, in which this is not feasible.

This is reflected in the fact that even the generified versions of the Java platform APIs which ship as part of J2SE 5.0 indeed expose raw types in their public interface. In the following, we will briefly describe a simple example of this phenomenon and also present an interesting type safe alternative provided by the designers of the new API. Our example is drawn from the `Collections` class and the relevant portions of the code are illustrated in Listing 2.11.

Listing 2.11: Raw types in the JDK

```
public class Collections {
    /* ... */

    public static final List EMPTY_LIST = new EmptyList();

    public static final <M> List<M> emptyList() {
        return (List<M>) EMPTY_LIST;
    }

    /* ... */
}
```

The field `Collections.EMPTY_LIST` shown in the example was already part of previous versions of the JDK and could therefore not be dropped in the new generified class. However, since the field is static, there is no way to parameterize it by a type variable, meaning that the field's type must be used in its raw form `List`. As assigning such a field to a parameterized version of a list used in some client code would always trigger an unchecked warning, a type safe alternative has been provided as part of the new API. In fact, using the polymorphic method `emptyList` illustrated in the example would not generate an unchecked warning when compiling the client code. Note, however, that this does not mean that the unchecked warning can really be avoided. Instead, if we look at the implementation of the method, we see that it uses a cast to the non-reifiable type `List<M>` which, in turn, would issue an unchecked warning. Here, it becomes evident that unchecked warnings are really tailor-made to flag *every* possibly unsafe operation and, therefore, they cannot be avoided when converting from a raw type to a parameterized type. However, what the addition of the method `emptyList` pretends to do is moving the unchecked warning from client code to the implementation of the JDK.

Chapter 3

Implementation

In the following sections, we will describe the actual implementation of the features discussed so far but also the things originally part of the semester project which could not be implemented completely during the given time frame.

Note that since JML builds on top of the MultiJava compiler and its utility classes, most modifications regard the MultiJava code as JML automatically benefits from them. Therefore, the actual modifications to JML are described separately towards the end of the chapter.

3.1 Support for JRE 5.0

A new version of the Java Platform usually also introduces a new JVM class file format which is adapted in order to accommodate new features such as additions to the Java language. Of course, a compiler written for a previous version of the platform would typically reject to compile against such a class file just as it would not be able to compile a source file of the new language.

However, a compiler written for a previous version of the Java language can often be easily adapted in order to be able to work with such class files by only extracting the information it knows and understands while simply ignoring the rest. Note that this is exactly what the Sun Java 5 compiler does if invoked through the command

```
javac -source 1.4
```

The design of the JVM class file format greatly simplifies the above process by providing a set of facilities which allow to attach *metadata* to classes and its members. In particular, the following simple constructs for adding information to classes, fields, and methods are supported:

- **Access and property flags:** This is a simple mask of *predefined* flags used to denote access permissions and various other properties of a class, field, or method. This bitmask e.g. reveals whether the type described by the class file in fact is a `class` or rather an `interface`, or whether a field has been declared to be `volatile`. Flags specific to methods include whether the method is `synchronized` or `native`.
- **Attributes:** Attributes provide a much more flexible way of adding metadata to a particular class, field, or method. In fact, an attribute consists of a name and an arbitrary string which represents the actual contents of the attribute. The JVM specification supports a set of *predefined* attributes by clearly stating how the contents of a specific attribute should be interpreted and what semantic information it contains. Currently, there are predefined attributes that can e.g. be used to mark a class member as being deprecated but also very familiar constructs such as the checked exceptions declared in the throws clause of a method are stored in special attributes in the class file. In addition, the JVM specification allows *user-defined* attributes to be stored in a class file as long as their names do not clash with those of predefined ones. The actual interpretation and handling of such attributes is then up to specific tools which recognize them.

The JVM specification expressly *requires* that a compiler or any other tool operating on the class file *silently* ignores access and property flags as well as attributes which may have been assigned special meaning in a *later* version of the JVM specification. This usually permits a compiler to work with such class files by safely skipping the extra information.

At its current version, the MultiJava compiler mainly targets version 1.4 of the Java language (despite some basic support for Java 5). In the following, we will describe how the compiler was adapted to make it cope with the class file format defined in J2SE 5.0. This makes it possible to run the MultiJava compiler as a Java *1.4* compiler on a Java *5* Platform which forms a first basic step to move the development and support of the compiler to the new platform. To that end, we will briefly describe how generics are mapped to the class file format. Please note that other Java 5 language additions – namely *varargs*, *enumerations*, and *annotations* – have also some minimal representation in a class file. However, no modifications were necessary to handle them and, therefore, they will not be discussed at this point. For a thorough discussion of all changes to the JVM class file format introduced by the Java 5 language additions [2], we refer the interested reader to the JVM specification which is currently only available as a final public draft version of JSR-202 [1].

3.1.1 Generics in a class file

Generic type information

As was discussed in 2.2, the JVM offers no runtime support for generics. However, the generic type information must still be stored in the class file as it is later needed by the compiler.

Prior to the support for generics, the type information of classes, fields, and methods was stored in so-called *descriptors*. A descriptor of classes and fields is just a string containing a simple encoding of the corresponding type while method descriptors similarly contain the types of the formal parameters and that of the return type. In order to preserve backward compatibility, the generic type information is *not* stored in those descriptors but, instead, a new predefined attribute is introduced – the so-called *Signature* attribute – which contains the generic type information of classes, fields, and methods.

Bridge methods

Under certain circumstances, generics require special methods – so-called *bridge methods* – to be inserted by the compiler. To see why, let us consider Listing 3.1.

Listing 3.1: Need for bridge methods

```
interface ValueHolder<T> {
    T getValue();
}

public class Implementor implements ValueHolder<String> {
    public String getValue() {
        return ”the value”;
    }
}
```

We see that the interface method implemented by the `Implementor` class must have `String` as its return type. However, assume the method is invoked on a variable whose static type is the interface, not the class. Since the JVM only knows about the erasure of the interface, the compiler would generate bytecode to invoke a method whose signature is the erasure of the interface's declared method, i.e.

Object getValue();

Since the JVM searches for methods by their *full* signature, i.e. also including the return type, no such method would be found at runtime on an instance of the implementing class as the return types differ. Therefore, the compiler must insert an appropriate bridge method into the `Implementor` class which simply delegates the call to the actual method specified in the code. Bridge methods are thereby explicitly marked as such by the special `ACC_BRIDGE` bit in their property flags.

3.1.2 Modifications to the compiler

Following the above explanations of how generics are mapped to the class file, it becomes clear that it was indeed very easy to adapt the MultiJava compiler to make it cope with the new class file format.

This was done by ensuring that for every field, method, and type declaration, the generic type information stored in the separate Signature attribute is only used by the compiler if generics are turned on. However, if the compiler is configured to only support version 1.4 of the Java language, the required type information is extracted from the type descriptor. This only required changes to the classes `CBinaryField`, `CBinaryMethod`, and `CBinaryClass` where the correct type signature is picked from the class file before passing it to the class file parser which then constructs an appropriate member based on the provided type information.

Bridge methods, on the other hand, need only be handled explicitly in that the compiler must be aware of the fact that they are synthetic methods generated by the compiler in order to e.g. ignore them during type checking. Since the MultiJava compiler already provided support for correctly handling synthetic methods, we merely had to mark bridge methods as being synthetic, which was done in the class `MethodInfo`.

3.2 Generics

3.2.1 Existing generics support

The work of this semester project largely builds on top of a recent master's thesis [5] which implemented an important part of generics for the MultiJava compiler. In this section, we will briefly discuss some aspects of that implementation and also describe the main classes at the core of the MultiJava compiler which will be referenced later on during the discussion of our implementation.

The MultiJava compiler uses two different abstractions for type declarations and for types used in the code. The abstract class `CClass` represents a type declaration and different specializations are provided which model types declared in a Java source file (`CSourceClass`) and types extracted from a class file (`CBinaryClass`). A `CClass` mainly serves as a data structure which holds the different type members such as fields and methods, while it also provides methods to lookup such members. The abstract class `CType`, on the other hand, stands for a type of the Java type hierarchy and it represents all the types used in the code, such as the types of fields and method parameters, but also the types to which expressions evaluate. A `CType` mainly contains the logic for checking the subtype relationship to other types which is required for verifying the correctness of e.g. assignments and method calls in the code. Additionally, it incorporates the type checking of the type itself which includes verifying whether a type declaration for the given type name really exists. As is done in Java, the MultiJava compiler mainly differentiates between primitive types and reference types, the latter being represented by the abstract class `CClassType`. Reference types defined by some type declaration are modeled by the class `CClassNameType`.

In [5], the MultiJava compiler has been extended to support generic types as well as polymorphic methods. Type variables introduced by generic type declarations are thereby represented by the special reference type `CTypeVariable` which mainly contains an array of types representing the upper bounds of the type variable. Parameterized types, on the other hand, have been implemented on top of `CClassNameType` by associating a set of actual type arguments to that type.

Since the MultiJava compiler lacks an abstraction for nested types, it was decided to store the type arguments in a two-dimensional array in which the type arguments of a nested type

HashMap<Integer, String>.HashIterator<Integer>

can be stored in the form (assuming that a type is simply represented by its name for illustration)

{ { "Integer", "String" }, { "Integer" } }.

The appropriate array structure containing the type arguments is constructed while parsing the type in the source file (or in the class file) and is then passed to the actual type. Finally, the erasure of a generic type is modeled separately by the special class `CErasedClassType`.

Despite the support for wildcards and raw types, the already existing generics implementation covered most other aspects of the generic type system. In particular, support for the following parts of generics was provided *prior* to this semester project:

- Generic type and method declarations, including type parameters with multiple upper bounds.
- Reading/Writing generic type signatures of fields, methods, and generic type declarations from/to class files (*not* including wildcard signatures, as described later in 3.2.2).
- Subtyping in the presence of type variables and parameterized types (*not* including the new subtyping facilities introduced by wildcards, as described in 2.3.1).
- Computing the erasure of a generic type.
- Inserting implicit generics casts where needed.
- Generating bridge methods where needed.
- Handling method resolution and method overriding in the presence of generic types.

As we can see, a very extensive generics support was already provided on which we could base our own implementation. However, a few *minor* details had to be adapted since at the time the existing implementation was elaborated, the specification of Java generics was only available as a public draft version of JSR-14 which slightly differs from the final specification given by the third edition of the Java Language Specification [9]. Examples of such changes include that it is now not permitted to instantiate arrays of non-reifiable types and that casts to non-reifiable types must issue an unchecked warning (see discussions in 2.2.1 and 2.2.2).

In addition to such changes, we have also tried to fix other problems we encountered during our work. For example, the existing implementation did not cope with the generic type signature of type parameters which have only interfaces in their bounds. In fact, the generic type signature of e.g. the type parameter `T extends Cloneable` would look like

```
T::Ljava/lang/Cloneable;
```

The general structure of such a type parameter signature is the specification of the type parameter's name followed by the individual upper bounds, each of them preceded by a colon. However, even if no class (as opposed to an interface) is specified as the first upper bound, an empty signature is nevertheless produced, hence the two subsequent colons in the signature above. The mishandling of such signatures was one of the reasons why the existing implementation could not be run on JRE 5.0.

3.2.2 Wildcards

In the following, the main aspects of adding support for wildcards to the MultiJava compiler will be described. We will thereby refer to the theory elaborated in 2.3 and discuss how the semantics of wildcards have been accommodated in the compiler.

Core wildcard support

A wildcard itself is represented by the new type `CWildcardType` which supports unbounded as well as upper and lower bounded wildcards. Wildcards can appear in a Java source file but must also be extracted from class files. In a class file, all the generic type information, including wildcards, is stored in so-called *signatures* which contain type information which is *not* part of the JVM type system but which is used by the compiler to reconstruct the generic type information originally specified in the source code. A signature contains a simple encoding of the type information where the different types are represented by a single character, possibly followed by some additional information such as the fully qualified name of a class type. Table 3.1 shows the encoding used for the individual Java types.

Encoding	Type	Encoding	Type
`B`	`byte`	`C`	`char`
`D`	`double`	`F`	`float`
`I`	`int`	`J`	`long`
`S`	`short`	`Z`	`boolean`
`L` *ClassType* `;`	class type	`[` *ArrayType*	array type
`T` *Identifier* `;`	type variable	`*`	unbounded wildcard
`+` *Bound*	upper bounded wildcard	`-` *Bound*	lower bounded wildcard

Table 3.1: Type signature encoding

For representing parameterized types, the type information about the actual type arguments is enclosed in angle brackets and immediately follows the encoding of the class name, so e.g. the type `List<? extends String>` would be encoded to the following signature:

```
Ljava/util/List<+Ljava/lang/String;>;
```

For being able to extract wildcards from the generic type signature of fields and methods, the class file parser of the MultiJava compiler was extended to correctly recognize the wildcard signatures shown in Table 3.1 and to build an appropriate instance of `CWildcardType` based on the information collected. On the other hand, *writing* such signatures to a class file during compilation for wildcards which appear in the source code is directly done in `CWildcardType`, thereby following the basic design of the MultiJava compiler.

Capture conversion

As was pointed out in 2.3.2, the process of capture conversion is an important aspect in the implementation of wildcards as it provides a convenient way of differentiating between wildcards which are not bound to any particular type and wildcards which are known to stand for some specific type. Such a distinction is very important since, depending on whether a wildcard is bound to some specific type or not, its semantic behavior is totally different.

Therefore, we have decided to strictly adhere to the implementation by capture conversion in that we separately model an *unbound* wildcard by the already introduced class `CWildcardType` while a *bound* wildcard captured during the process of capture conversion is represented by the new class `CCaptureType`. The latter largely behaves like a normal type variable and, as such, is implemented as a subtype of `CTypeVariable`.

Capture conversion is performed on the type of every expression which is evaluated. If that type is a parameterized type, all top-level wildcard type arguments (`CWildcardType`) are transformed into fresh type variables (`CCaptureType`) with appropriate upper bounds and a lower bound. The actual logic for this transformation is implemented in the `CCaptureType` class itself which provides a static factory method which creates such a type variable from a given wildcard following the definition given in 2.3.2. Note that for the transformation, the formal type parameter of the

wildcard is required. Since the parameterized type containing that wildcard always knows what the corresponding type parameter is, the type parameter is associated to the wildcard during the type checking of the parameterized type. This ensures that the necessary information is always available when performing capture conversion.

We have already mentioned that, once an instance of `CCaptureType` has been created with the appropriate bounds, it behaves like any other type variable explicitly declared in the code, at least in most situations. In fact, only when assigning a value to them, the two kinds of type variables behave differently: while a normal type variable can only be assigned a value of its own type, a type variable generated during capture conversion relaxes this requirement by also allowing values to be assigned to it if those values are assignable to its lower bound.

Type containment

In 2.3.1, a special relationship between type arguments called *type argument containment* was introduced which defines the subtype relationships between parameterized types belonging to the same generic type declaration.

In our implementation, checking whether a type argument A contains another type argument is done in the type A itself. This is easily possible since checking the containment relationship between two type arguments is totally independent of other type information of the parameterized types in which the type arguments are used. Note that the actual type containment check is only interesting in the presence of wildcards, since, for all other types, it reduces to a mere type equality check. For wildcards, the type containment rules presented in 2.3.1 have been implemented in the `CWildcardType` class.

3.2.3 Raw types

In the following, the main aspects of adding support for raw types to the MultiJava compiler will be described. We will thereby mainly discuss the implementation of the features described in 2.4 but also mention some of the issues encountered during the implementation.

Type arguments for raw types

In the existing implementation of parameterized types, many places in the code relied on the fact that every type belonging to a generic type declaration has appropriate actual type arguments for every corresponding type parameter. Of course, this is a more than valid assumption but only in the absence of raw types. However, it is actually possible to synthetically equip raw types with special type arguments which basically cover their semantics, namely the erasures of the type parameters defined in the corresponding generic type declaration.

In our implementation, these special type arguments are attached to a raw type during its type checking. This ensures that, even in the presence of raw types, the old assumption of all generic types having appropriate type arguments still holds. In addition, treating a raw type similarly to any other parameterized type permits to naturally represent the erasure of a generic type by such a raw type instead of using the special purpose class `CErasedClassType` as was done before. Of course, this does *not* mean that raw types do not require special handling in the code. In fact, class member accesses and assignments involving raw types (both discussed below) still need to be handled explicitly. However, this only requires punctual changes in the code.

Type checking

The additional type checking required for supporting raw types mainly consists of two parts: on the one hand, assignments between parameterized types and raw types must be handled correctly while, on the other hand, the impact of accessing a class member on a raw type must be considered. Listing 3.2 shows an example which illustrates both situations.

Listing 3.2: Raw type invocation

```
class Super<T> {
    public List<T> superList;
}

public class Sub extends Super {
    public static void main(String[] args) {
        Super<String> sup = new Sub(); // Unchecked warning

        Sub nonRaw = new Sub();
        nonRaw.superList = new List(); // Unchecked warning
    }
}
```

For assignments, we have to check whether we are assigning a raw type to a parameterized type in order to eventually issue an unchecked warning. As we can see in the example, it is not always immediately apparent in the code whether this is the case or not. In fact, in our sample assignment, `Sub` is not a raw type, but its supertype `Super` is a raw type, so we must issue an unchecked warning. More generally, what we have to check is whether the value being assigned is a raw type invocation of the target type of the assignment. That check is performed on every valid assignment and eventually produces an unchecked warning. An equivalent check is performed for every actual type argument passed to a method or constructor, since passing an argument is semantically equivalent to a normal assignment.

For member accesses, the situation is very similar in that we must also be able to check whether the type on which the member is accessed is a raw type invocation of the type where the member is actually declared. In our example, once more, we see that we are accessing a field on the non-raw-type `Sub`, but the field is declared in `Super`, which, in our case, is a raw type. Therefore, we must again issue an unchecked warning, since we are writing a field which is accessed on a raw type and whose type changes by erasure. More generally, whenever we identify a field or a method being accessed on a raw type, we apply the rules described in 2.4.2 to figure out whether an unchecked warning must be generated or not. In addition, the type of the member access expression is adapted accordingly by eventually replacing it by its own erasure, unless the member is declared to be static.

As we can see, checking assignments and member accesses is both based on the ability to check whether a type is a raw type invocation of some given generic type declaration. Since this functionality is required at different places, we have integrated it directly into the class `CClassType` which is the base class for reference types. The actual implementation takes an object representing a type declaration (`CClass`) as its sole argument and returns whether the type on which the method is invoked is a raw type invocation of that type declaration. This is simply done by going up the supertype hierarchy of the type until we get to the appropriate class and then checking whether that supertype is a raw type.

Unchecked warnings

During compilation, the MultiJava compiler provides feedback to the user not only by reporting eventual compilation errors but also by displaying different types of warnings or even more general informational messages. The compiler thereby provides appropriate compiler switches to control which of these messages to report to the user while suppressing all others. The MultiJava compiler even supports custom filters which can be loaded via reflection to provide an even more fine-grained control over the display of such messages. Either type of message filtering relies on a proper classification of the individual messages which have a type associated to them which tells the compiler what kind of message it is. Examples of such message types are simple *errors* but also different flavors of *warnings* (depending on their severity) are supported.

Since unchecked warnings are a very special kind of warnings, it was decided to introduce a separate type of message for them and to add a specific compiler switch for controlling the display of such warnings. Following the behavior of the Sun Java compiler, by default, unchecked warnings are not reported to the user, but they can be turned on using the custom compiler switch `--Xlint` as in

```
java org.multijava.mjc.Main --Xlint unchecked
```

Note thereby that even though unchecked warnings as such are not reported to the user if not explicitly enabled, the mere fact that such warnings occurred during compilation is nevertheless reported since it is usually important for the user to be aware of the presence of unchecked warnings.

Note also that the annotation facilities introduced in Java 5 allow unchecked warnings to be suppressed in specific parts of the code only, supported by the special annotation

```
@SuppressWarnings("unchecked")
```

While such annotations are not yet supported in the MultiJava compiler, the fact of having a special message type for unchecked warnings should facilitate their suppression in the scope of such an annotation.

3.2.4 Limitations of the generics support

The MultiJava compiler is now able to work with class files part of JDK 5.0. However, due to some punctual bugs in the compiler, the generics support is not yet robust enough to correctly handle all the generic features in the Java 5 platform APIs, meaning that some correct constructs are rejected by the MultiJava compiler. As a concrete example, it is currently not possible to reference a type parameter in its own upper bound even though this is allowed by the Java Language Specification. In addition, some generic constructs – such as generic arrays – are still not correctly extracted from a class file. In order to facilitate future development on the generics support in the MultiJava compiler, we will write appropriate bug reports[1] which allow to reproduce the still persisting problems we have encountered during our work.

Unfortunately, the above mentioned bugs currently prevent the MultiJava compiler to use the generic type information contained in class files part of the JDK since those classes contain constructs which would lead to spurious type checking errors during compilation. For that reason, JDK classes are currently treated specially in that their generic type information is *not* parsed, meaning that generics can currently only be used in the client code itself. This temporary limitation is implemented in the class `org.multijava.mjc.Main` and can be removed as soon as the above bugs have been fixed.

3.3 Testcases

A number of simple testcases have been created in the `org.multijava.mjc.testcase.java5` package and integrated into the MultiJava testing framework to test the individual aspects of our implementation. Every unit test consists of a single Java source file together with a text file which contains the messages the compiler is expected to produce upon processing the source file. This expected compiler output is then compared to the actual one during the execution of the unit test.

For wildcards, the main aspects covered by the unit tests are checking the validity of wildcard type arguments, the subtyping among wildcard-parameterized types, the process of capture conversion, and assignments involving (captured) wildcards. For raw types, mainly the member accesses on raw types as well as assignments involving raw types are tested.

Interestingly, while writing the unit tests, we have discovered several bugs in the Sun Java 5 compiler (and also in the Eclipse compiler). Some of them turned out to be fixed but only in the current beta version of the Java 6 compiler. A totally new bug was also found[2]. In our opinion,

[1] see http://sourceforge.net/projects/multijava

[2] see http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6452148

this reflects the fact that the generic type system in Java indeed contains many subtleties which are not totally apparent at first sight. An interesting consequence of this fact we have encountered during the elaboration of this semester project is that many reports for bugs in the generic type system are characterized by an extensive discussion which tries to clarify whether it really is a bug or not.

3.4 Modifications to the JML tools

As was already mentioned, the actual logic of our implementation resides in the MultiJava compiler since JML builds on top of it and therefore automatically benefits from the modifications to the MultiJava code. Hence, only some very basic changes to the JML tools [4] were necessary. More precisely, the *JML checker* and the *JML compiler* have been adapted to allow them to take advantage of all the generics support provided by the MultiJava compiler.

3.4.1 JML checker

The `jml` (`org.jmlspecs.checker.Main`) tool is known as *JML checker* and it provides the most basic tool support for JML by simply parsing and typechecking the JML specifications. This already ensures that the specifications are at least syntactically correct and that they e.g. do not contain references to non-existing types or variables. However, the JML checker does not generate code of any kind.

In order to make the JML checker cope with generics, only a few very basic changes were necessary. In particular, we have adapted the JML parser to make it recognize the type parameter definitions in generic type declarations (`JmlTypeDeclaration`) and polymorphic method declarations (`JmlMethodDeclaration`). Since the actual implementation of the JML classes for generic type and method declarations are based on their MultiJava counterpart, i.e. `JTypeDeclaration` and `JMethodDeclaration`, respectively, the newly parsed type parameters could basically be passed to the MultiJava classes which already handled them correctly. This already allowed us to run the JML checker on a Java source file containing generic type information. As we can see in Listing 3.3, generic types can also be used in the JML specifications.

Listing 3.3: JML checker example

```
public interface List<E> extends Collection<E> {
    /*@ pure @*/ int size();

    /*@ pure @*/ E get(int index);

    /*@ also
      @ ensures \result
      @     <==> o instanceof List
      @         && (size() == ((List<E>) o).size())
      @         && (\forall int i; 0 <= i && i < size();
      @                 (get(i) == null && ((List<E>) o).get(i) == null)
      @                 || get(i).equals(((List<E>) o).get(i)));
      @*/
    /*@ pure @*/ boolean equals(/*@ nullable @*/ Object o);
}
```

The example mainly illustrates a possible JML specification for the `equals` method of a list. We see that generic types cannot only be used in the Java code but also in JML specifications, where we automatically benefit from all the typechecking implemented in the MultiJava compiler. As an example, the above casts to `List<E>` in the JML specification would indeed generate unchecked warnings, just as we would expect.

3.4.2 JML compiler

The `jmlc` (`org.jmlspecs.jmlrac.Main`) tool is known as *JML compiler* and it provides one way of checking the correctness of JML specifications by runtime assertion checking, i.e. by producing code for the JML assertions which checks their correctness at runtime.

The JML compiler employs a special compilation strategy called *double-round compilation* which is described in [6]. In essence, the compiler first runs the JML checker on the original source file containing the Java code as well as the JML specifications. It then generates runtime assertion checking code for the JML specifications and adds appropriate nodes representing that code to the abstract syntax tree. The code represented by the modified abstract syntax tree is then written to a temporary file and compiled in a second run using the MultiJava compiler, which finally produces the actual bytecode containing both the original Java code as well as the runtime assertion code for dynamically checking the JML specifications.

Unfortunately, the concrete implementation of this compilation strategy turned out to lead to several problems in conjunction with generics. One of the main issues encountered is caused by the way the generated assertion checking code is inserted into the original code. More concretely, as described in [6], the JML compiler generates special assertion methods which contain the code to check different JML specifications such as pre- and postconditions or class invariants. If the type being annotated is a class, these assertion methods can be directly added to the class itself. For interface specifications, however, the interface cannot directly host those methods since all methods in an interface must be abstract. Therefore, the solution adopted in the JML compiler is to add an *inner* class to the interface to which the assertion methods can then be added. However, the context of inner classes inside an interface is always *static*, meaning that possible type parameters of the interface *cannot* be accessed from within this context. In particular, this means that generic interface declarations would break the compilation since the interface's type parameters usually need to be referenced by the generated assertion methods.

In order to circumvent this problem, we have decided to drop all the generic type information just before starting the second compilation run by only writing the erasure of all generic types to the temporary file. This clearly solves the above problem and allows the JML compiler to be run on source files containing generic types both in Java code and in JML specifications. However, even though all the typechecking on the original source file is performed including the generic type information, the obvious drawback is that *no* generic type information is included in the generated class file since the actual compilation to bytecode is done in the second compilation run in which no generic type information is available anymore.

Chapter 4

Conclusion and future work

4.1 Conclusion

In this report, a rough overview of the parametric polymorphism as realized in the Java programming language was given. Thereby, a special emphasis was placed on discussing many of the subtleties of the generic type system and how they relate to the certainly most crucial and controversial design decision of not providing runtime support for Java generics. In addition, a thorough treatment of the semantics of wildcards and raw types was provided and the implementation of those special features on top of the MultiJava compiler was described.

4.2 Generics

The MultiJava compiler now supports most features of Java generics. However, an important major aspect still missing is a type inference algorithm for polymorphic methods. In most cases, such an algorithm allows to automatically infer the types of method type variables without requiring the programmer to explicitly specify them for every polymorphic method call. The algorithm is fully specified in the Java Language Specification [9] but its implementation seems to be nevertheless involved since some features of the type system such as wildcards and intersection types make the type inference algorithm more powerful but at the same time also inherently complex. In particular, in the presence of wildcards, there is not always a unique optimal type which can be inferred from a given set of actual type parameters, as described in [13].

4.3 Java 5 support

Beside generics, J2SE 5.0 introduced further significant additions to the Java programming language whose specification and development was driven by two different JSRs. JSR-201 embodies several language additions which all aim at facilitating the development of Java programs by providing built-in support for several common programming idioms. Most of these features are mere convenience constructs to reduce the clutter in the code but a very powerful support for typesafe *enumerations* is also added to the language. JSR-175, on the other hand, specifies an annotation facility which allows to associate *metadata* with Java code. This avoids having to maintain separate files in which the metadata is stored and, more importantly, the annotations can also be retained in the bytecode and queried by tools via reflection.

In this semester project, we have adapted the frontend of the MultiJava and JML compilers to support the syntax of these language additions. However, the actual semantics need still be implemented.

4.4 Generic Universes

The Universe type system [12] is an ownership type system which hierarchically partitions the object store into so-called *universes* while providing control over references between different universes. Compared to other ownership type systems, the Universe type system is mainly characterized by its simplicity and ease of use.

The Universe type system was implemented on top of the MultiJava compiler and the JML tools. A proposal for integrating the type system into the generics mechanism of Java 5 is described in [8] which allows the Universe type system to benefit from the improved static type checking facilities provided by Java generics. The extension of the Universe type system to support generics was originally part of this semester project but could unfortunately not be realized in the given time frame.

Bibliography

[1] JSR-202: Java class file specification update. http://www.jcp.org/en/jsr/detail?id=202.

[2] New features and enhancements in J2SE 5.0. http://java.sun.com/j2se/1.5.0/docs/relnotes/features.html.

[3] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the java programming language. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 183–200, New York, NY, USA, 1998. ACM Press.

[4] Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. In *International Journal on Software Tools for Technology Transfer, volume 7, number 3*, pages 212–232, June 2005.

[5] Tongje Chen. Extending MultiJava with generics. Master's thesis, Iowa State University, 2004.

[6] Yoonsik Cheon. *A Runtime Assertion Checker for the Java Modeling Language*. PhD thesis, Department of Computer Science, Iowa State University, Ames, April 2003.

[7] 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.

[8] Werner Dietl, Sophia Drossopoulou, and Peter Müller. Generic Universe Types. Technical report, to appear.

[9] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley Professional, June 2005.

[10] Andrew Kennedy and Don Syme. Design and implementation of generics for the .NET common language runtime. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 1–12, New York, NY, USA, 2001. ACM Press.

[11] Gary T. Leavens and Yoonsik Cheon. Design by contract with JML. 2006.

[12] Peter Müller and Arnd Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical Report 279, Fernuniversität Hagen, 2001.

[13] Mads Torgersen, Erik Ernst, Christian Plesner Hansen, Peter von der Ahé, Gilad Bracha, and Neal Gafter. Adding wildcards to the java programming language. In *Journal of Object Technology, vol. 3, no. 11*, pages 97–116, December 2004.

[14] Mads Torgersen, Christian Plesner Hansen, Erik Ernst, Peter von der Ahé, Gilad Bracha, and Neal Gafter. Adding wildcards to the java programming language. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 1289–1296, New York, NY, USA, 2004. ACM Press.