Bytecode Support for the Universe Type System

Alex Suzuki

Semester Project Report

Software Component Technology Group
Department of Computer Science
ETH Zurich

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

WS 2004/2005

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

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Software Component Technology Group

Abstract

The goal of this semester project is to extend the MultiJava compiler to emit extensions used in the Universe type system into the generated Java class file. This involves finding a suitable binary representation for the extensions, and a means of storage which does not conflict with the Java class file format. This document presents a binary encoding of the Universe modifiers, and two possible approaches to their storage in the class file. Chapter 1 serves as an introduction to the problem of aliasing in object-oriented programming and gives a short introduction to the Universe type system. Chapter 2 gives an overview of the Java class file format and a quick introduction to the new J2SE 5.0 annotation facility. Chapter 3 provides a detailed look at our two implementation approaches, while chapter 4 provides an example. Chapter 5 describes the changes and extensions made to the MultiJava compiler. Chapter 6 then presents conclusions and possibilities for future work.

Contents

Chapter 1

Introduction

1.1 Aliasing in object-oriented programming

In object-oriented programming, sharing mutable objects is natural and often unavoidable. Much of the efficiency of object-oriented programming stems from *aliasing*, object cloning being an expensive alternative. An object is said to be *aliased* if it is being referenced by multiple other objects. Unfortunately, uncontrolled aliasing can be dangerous. Outside objects may hold references to objects representing some fragile composite structure (like a syntax tree), and may thus bypass the structure's well-defined interface, in the worst case destroying the structure. This is known as *representation exposure*.

One example of representation exposure is a security breach[1] in JDK 1.1.1 ([1] describe this in greater detail) that allowed digitally signed code to be manipulated so that it appeared to be signed by another signer in the list of valid signers known to the Java runtime. Through the method `Class.getSigners()` it was possible to obtain a reference to the array of signers of a class. Since arrays are mutable objects and the list of trusted signers was available, a trusted signer could simply be inserted into the array of signers.

1.2 The Universe type system

The Universe type system, as described in [11], is a type system designed to control aliasing and dependencies. It does so by hierarchically partitioning the object store into so-called *universes* and enforcing an ownership model on the objects. The basic idea is to distinguish between an object X and its *representation*, the objects owned by X. The objects owned by X are known to be in X's universe. These objects implement the behavior of X. Every object has a unique owner, except for the *root* objects which have no owner and belong to the *root universe*.

The key idea is that the Universe type system does not allow read-write references to cross universe boundaries. X is allowed to hold a read-write reference to Y only if X and Y are in the same universe (i.e. they share the same owner) or if Y is owned by X. However, certain implementation patterns are made impossible by these restrictions. For instance, think of a linked-list with a List object owning the list nodes (the nodes form the list's representation), and an iterator which cycles through the linked nodes. In this scenario, the iterator is not allowed to hold references to the list's nodes, since they are neither in the same universe nor does the iterator own the nodes. The situation is depicted in figure 1.1.

To add flexibility, the Universe type system adds *read-only* references, which may refer to objects in arbitrary universes, but which disallow modification of object state or calls to non-pure methods, meaning that they cannot be used to perform field updates or invoke methods with side-effects.

[1] This bug was fixed in the next version, JDK 1.1.2

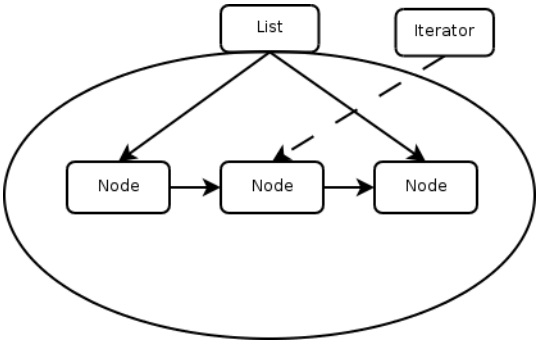

Figure 1.1: Linked list with nodes and iterator. The iterator is not allowed to hold a read-write reference to the nodes.

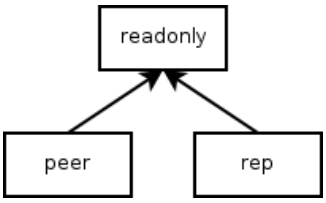

Figure 1.2: Type hierarchy for simple references.

1.3 Classification of references

The classification of references is done by adding a modifier to the reference. Normal references (i.e. references to objects that have the same owner as `this`) are of type `peer`, references from an object X to objects in the universe of X (i.e. objects owned by X) are of type `rep`, and read-only references, which may point anywhere, are of type `readonly`. A type hierarchy can be imposed on the references, following from the intuition that a `peer` or a `rep` reference may be assigned to a `readonly` reference, and that `peer` and `rep` references are not assignable to each other. Assigning a `rep` reference to a `peer` variable would mean that the `peer` reference crosses universe boundaries, which is forbidden. Vice versa the `rep` reference would point to an object not owned by `this`. The type hierarchy for simple reference types is shown in figure 1.2. Note that `readonly`, `peer` and `rep` are attributes of the reference, not of the object. The same object can be referenced as `peer` from objects in the same universe, as `rep` from its owner and as `readonly` from anywhere.

Arrays of primitive types behave exactly the same way, but arrays of reference types are slightly more complicated, because not only the array object may have a modifier, but also the elements. The type hierarchy for array types is shown in figure 1.3. Arrays with `rep` element annotation are forbidden, since the array does not have its own universe.

1.4 Bytecode support

Prior to this project, the existing compiler correctly processed Java source files with universe keywords. However, the class file it generated as a result of the compilation did not include universe modifiers and method purity. That information was simply lost. As a consequence, source code for all classes was necessary to properly typecheck a program composed of multiple classes. This assumption is unrealistic, in many cases source code is not available, the standard library being a prominent example. References in compiled classes were simply interpreted as being implicitly `peer`, and methods as being non-pure.

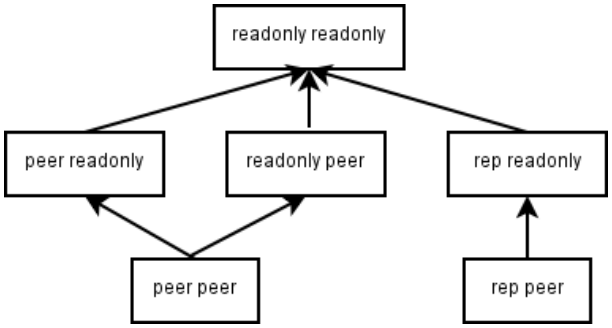

Figure 1.3: Type hierarchy for arrays of reference types.

For instance in the previously mentioned list and iterator example, depicted in figure 1.1, the iterator has a field of type `readonly` Node holding a reference to the current node. The field has the `readonly` modifier so that clients cannot change the list's representation through the iterator. If the source file containing the code for the iterator class would then be compiled to a class file, the `readonly` modifier of the field would be lost. If the source file is then made unavailable to us, we have no way of figuring out that the type of the field is `readonly` Node and not just Node, we would have to treat it like any other Java reference. Modifying the state of the referenced Node object would then be possible (assuming it is visible), and would not generate a type error.

The goal of this semester project was to emit the universe modifiers into the class file in a way that conforms to the Java class file format, so any standards-conforming Java virtual machine implementation would still be able to understand the class file. With the universe information stored in the class file, source code for all classes is no longer needed, as long as the classes were compiled using our proposed extensions.

Chapter 2

The class file format

Most of the information presented in this chapter is taken from the Java Virtual Machine specification [10], some of it is at current date only available in the maintenance releases[1].

2.1 Structure of a `class` file

A `class` file consists of a stream of 8-bit bytes. All 16-bit, 32-bit and 64-bit quantities are constructed from reading in two, four, and eight consecutive 8-bit bytes, respectively. Multibyte data items are always stored in big-endian order, where the high bytes come first. We use a C-like structure notation, identical to the one used in [10], to describe the class file format. The types `u1`, `u2`, and `u4` represent an unsigned one-, two-, or four-byte quantity, respectively. A class file consists of a single `ClassFile` structure:

```
ClassFile {
    u4 magic;
    u2 minor_version;
    u2 major_version;
    u2 constant_pool_count;
    cp_info constant_pool[constant_pool_count-1];
    u2 access_flags;
    u2 this_class;
    u2 super_class;
    u2 interfaces_count;
    u2 interfaces[interfaces_count];
    u2 fields_count;
    field_info fields[fields_count];
    u2 methods_count;
    method_info methods[methods_count];
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

2.1.1 Classes

Every class file contains the definition of a single class or interface. Its visibility is stored as a bitmask in the `access_flags` item. Its name and the name of its super class are given by the `this_class` and `super_class` items, which are indexes into the constant pool. The constant pool entries at those indexes are strings holding the actual names. The implemented interfaces are stored in the `interfaces` array, which again contains indexes into the constant pool. For every

[1] See http://java.sun.com/docs/books/vmspec/2nd-edition/jvms-maintenance.html

field and every method there exists a `field_info` and a `method_info` structure, respectively, which we will describe in the next two sections.

In addition, a class may have a variable number of *attributes* (in this case class attributes) associated with it. One example of a class attribute is the `SourceFile` attribute, which stores the name of the source file from which this class was compiled[2]. Attributes are crucial for our implementation, and are discussed later on in greater detail.

2.1.2 Fields

Every field is described by a `field_info` structure. The format of this structure is:

```
field_info {
    u2 access_flags;
    u2 name_index;
    u2 descriptor_index;
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

`access_flags` is a mask of flags used to denote access permission (e.g. `private`, `public`, etc.) and properties of the field (e.g. `static`, `final`). The name of the field is described by `name_index` which is an index into the constant pool, and the type of the field is given by `descriptor_index` which also points to an entry in the constant pool.

Every field can be annotated by a variable number of attributes. One example of a field attribute is the `ConstantValue` attribute, which represents the value of a constant field.

2.1.3 Methods

Every method, including constructors and the class or interface initialization method, is described by a `method_info` structure. The structure has the following format:

```
method_info {
    u2 access_flags;
    u2 name_index;
    u2 descriptor_index;
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

`access_flags` and `name_index` serve the same purpose as in the `field_info` structure. For methods, `descriptor_index` yields the methods signature (i.e. its return and parameter types) in a textual form. A method returning no value and accepting two parameters, one `int` and one `Object`, will have the following signature: `(ILjava/lang/Object;)V` (I = Integer, L = reference type, V = void)

Like fields, methods can be annotated by a variable number of attributes. An example of a method attribute is the `Exceptions` attribute which describes the exceptions a method may throw.

2.2 Attributes

Attributes are a way of adding additional information to fields, methods, classes and even attributes themselves. They serve as a way to extend the *metadata* of a class. The same concept can be found in other languages and platforms, the *.NET* platform being a prominent example. There are several pre-defined attributes, like for instance the SourceFile attribute (which gives the source filename of the compiled class) or the ConstantValue attribute, which associates a constant value to a static field.

[2] The filename does not have to be the same as the name of the class

2.2.1 Structure of an attribute

All attributes have the following general format:

```
attribute_info {
    u2 attribute_name_index;
    u4 attribute_length;
    u1 info[attribute_length];
}
```

`attribute_name_index` points to a constant pool entry which holds a string representing the name of the attribute, e.g. "SourceFile". The `attribute_length` item specifies the length of the subsequent information in bytes. The `info` item may contain arbitrary information, its length is limited by `attribute_length`.

2.2.2 Defining custom attributes

Compilers may emit class files containing new attributes. Java virtual machine implementations are permitted to recognize and use these new attributes, and are required to silently ignore attributes they do not recognize. An example use of custom attributes would be to include vendor-specific debugging information. Attribute names follow the same convention as package names in Java, e.g. `org.multijava.universe_field` would be a valid attribute name.

2.3 Annotations

As of J2SE 5.0, Java includes *annotations*, a new metadata facility to annotate programs. Annotations are described in detail in [7], only a short overview is provided here. Good tutorial introductions are given in [4] and [5].

2.3.1 Introduction

Annotations are a new way of adding metadata to program elements such as fields, methods, parameters and so on. So conceptually they are not much different from attributes, which we described in the earlier sections. However, unlike attributes, annotations are not a low-level concept, they are fully embedded into the core Java language. Annotations are *typed* (i.e. it is necessary to define an *annotation type*) and can be used directly in Java source code.

Development tools can read these annotations and use them to perform various tasks or to justify assumptions. For instance, a stub generator could generate remote procedure calls for methods that have an annotation which marks them as being designed for remote use. Another advantage of choosing annotations over attributes is that they are exposed through the Java Reflection API (`java.lang.reflect`), while attributes are not.

2.3.2 Annotation types

In order to annotate a program element, one needs to first define an annotation type. Annotation types are reminisicent of Java interfaces, and are declared in much the same way. An annotation type for an annotation that marks an element as being unsafe could be defined like this:

```
public @interface Unsafe {
    String reason();     // the reason why it is unsafe
    int    bugsFiled();  // how many bugs have been filed against this element
}
```

Methods in annotation types are only allowed to return primitive types, `Class`, `String`, enum types, annotation types and arrays of the preceding types. It is also possible to assign default values to methods.

2.3.3 Annotating program elements

Program elements can be annotated by specifying the name of the annotation type, and initializing its members, before the program element to be annotated. In the example a field is annotated with the `Unsafe` annotation.

```
public class C {
    @Unsafe(reason="This method behaves dangerously", bugsFiled=42)
    public void m() {
        // bad things happen here
    }
}
```

2.3.4 How annotations are stored in the class file

Annotation types are stored in the same way as regular Java interfaces, in a class file that has the same name as the annotation type. The annotations themselves are stored in attributes, which we will describe in the following sections. Much of the information is taken from the revised Java Virtual Machine specification [10].

The RuntimeVisibleAnnotations attribute

The RuntimeVisibleAnnotations is a variable-length attribute that can occur in the `field_info`, `method_info` and `ClassFile` structures. It contains the run-time visible annotations of the associated field, method or class. There is also an attribute RuntimeInvisibleAnnotations which stores the annotations which are not exposed to reflective APIs, however it is not of interest to us, because one of the reasons we would like to use annotations is exactly because they can be queried at run-time using the Reflection API. Its format is identical to the RuntimeVisibleAnnotations attribute, which has the following form:

```
RuntimeVisibleAnnotations_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 num_annotations;
    annotation annotations[num_annotations];
}
```

`attribute_name_index` refers to a constant pool entry holding the string "RuntimeVisibleAnnotations". `attribute_length` is given by $2+num\_annotations*sizeof(annotation)$, where `num_annotations` is the number of (run-time visible) annotations assocated with the attributed element. The actual annotations are stored in the `annotations` array, every entry being of the following form:

```
annotation {
    u2 type_index;
    u2 num_element_value_pairs;
    { u2 element_name_index;
      element_value value;
    } element_value_pairs[num_element_value_pairs];
}
```

An `annotation` consists of the name of the annotation type, which is stored in the constant pool entry with index `type_index`, and a number of $(name, value)$ pairs. The name of each pair corresponds directly to the name of the method in the annotation type. The value of the pair is a discriminated union. The possible data members of the union are directly related to the possible return types of the corresponding method in the annotation type. Every value has the following format:

```
element_value {
    u1 tag; // discriminator
    union {
        u2 const_value_index;
        {
            u2 type_name_index;
            u2 const_name_index;
        } enum_const_value;
        u2 class_info_index;
        annotation annotation_value;
        {
            u2 num_values;
            element_value values[num_values];
        } array_value;
    } value;
}
```

The union is discriminated by the `tag` item. Its possible values and their interpretation are given in table 2.1. For primitive and String types, the `const_value_index` item is used, the constant pool entry at that index giving the value. For enums, (nested) annotations, arrays and Class types, constant pool entries are used as well.

Tag value	Element type	Tag value	Element type
B	byte	C	char
D	double	F	float
I	int	J	long
S	short	Z	boolean
s	String	e	enum constant
@	annotation type	[	array
c	Class		

Table 2.1: Tag values and their corresponding types

The RuntimeVisibleParameterAnnotations attribute

For methods, a special attribute RuntimeVisibleParameterAnnotations exists that allows individual parameters to be annotated. The format is almost the same as the RuntimeVisibleAnnotations attribute, except that not just one list of annotations is stored, but one list for every parameter:

```
RuntimeVisibleParameterAnnotations_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u1 num_parameters;
    {
        u2 num_annotations;
        annotation annotations[num_annotations];
    } parameter_annotations[num_parameters];
}
```

The `num_parameters` byte stores the number of parameters[3]. Again, there exists a related attribute RuntimeInvisibleParameterAnnotations with the obvious meaning.

[3]This actually duplicates information already available in the method descriptor, see 2.1.3

Chapter 3

Implementation

3.1 Encoding of universe modifiers

First we specifiy a binary encoding for universe modifiers. For simple reference types, we have four possibilities: a reference may be either `peer`, `rep`, `readonly` or may have no annotation. The latter case only occurs when a method has a parameter of primitive type, the byte is then used as a placeholder. We use a straight-forward 2-bit scheme, shown in table 3.1 to encode the universe modifier of a simple reference type.

modifier	binary encoding
(none)	00
`peer`	01
`rep`	10
`readonly`	11

Table 3.1: Binary encoding for simple reference types

Arrays of reference type have universee modifiers for both the elements and the array itself. For instance, it is possible to define an array like this: `rep peer` Object[] ar; In this case the array is owned by `this`, and its elements may refer to objects in the same context as the array. Some combinations are not possible, becuase array elements are not allowed to have the `rep` modifier. So to encode the modifiers of an array type, 4 bits are sufficient. We just reuse the encoding scheme for simple reference types, and use the low 2 bits to encode the modifier of the array object, and the subsequent 2 bits to encode the modifier of the array elements, as shown in table 3.2.

array modifier	binary encoding
`peer peer`	0101
`peer readonly`	1101
`rep peer`	0110
`rep readonly`	1110
`readonly peer`	0111
`readonly readonly`	1111

Table 3.2: Binary encoding for array types

So one byte is sufficient to encode the universe modifiers of a simple reference or array type. In the actual implementations we use the highest bit to indicate if the reference is an array. So `rep peer Object[] ar;` becomes `10000110 = 0x86`.

Method purity is encoded trivially with a single bit, as shown in table 3.3.

method purity	binary encoding
`pure`	1
(non-pure)	0

Table 3.3: Binary encoding for method purity

3.2 Implementation with simple attributes

This approach uses class, field and method attributes, as discussed in 2.2, to store universe modifiers and method purity.

3.2.1 Classes

To enable future extensions and changes to this first encoding, we record the version of the encoding used. We also include a flag to indicate if the class has been compiled with run-time support enabled. We store the version string (e.g. "1.0") and the boolean flag in a class attribute. The attribute has the following form:

```
universe_class {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 universe_version_index; // version of the encoding
    u1 runtime_support;       // indicates  if  run-time checks are  enabled
}
```

`attribute_name_index` points to a constant pool entry holding the string "org.multijava.universe_class". The class version is stored as a string in the constant pool, and `universe_version_index` points to it. The boolean flag that indicates if the class has run-time support enabled is stored in the `runtime_support` byte. The length of the attribute is always equal to 3.

Run-time support in the context of the Universe type system comprises checking casts of reference types, and assignment of array elements. The mechanisms are illustrated in [12].

3.2.2 Fields

For a field of a reference type, we need to store the universe modifier of the field's type. We use a field attribute of the following form to store an encoded universe modifier using the scheme described in 3.1.

```
universe_field {
    u2 attribute_name_index;
    u4 attribute_length;
    u1 universe_modifier; // encoded universe modifier
}
```

The attribute's length (stored in `attribute_length`) is always one, since one byte is sufficient to store the universe modifier for any type. `attribute_name_index` points to a constant pool entry holding the string "org.multijava.universe_field". The `universe_modifier` byte stores the encoded universe type modifier.

3.2.3 Methods

For every method, we need to store if the method is pure, the universe modifiers of the method's parameters, and the modifier of the return type. We use a method attribute of the following form to store this information:

```
universe_method {
    u2 attribute_name_index;
    u4 attribute_length;
    u1 purity;                              // the method's purity
    u1 return_modifier;                     // return type modifier
    u1 param_modifiers[attribute_length - 2]; // parameter modifiers
}
```

The attribute's length depends on the number of parameters, and is given by $2 + num\_params$. `attribute_name_index` points to a constant pool entry holding the string "org.multijava.universe_-method". The `purity` byte indicates if the method is pure, `return_modifier` holds the universe modifier of the return type, and `param_modifiers` are the universe modifiers for each parameter.

To simplify parsing, methods that do not return a value (i.e. have `void` return type) have a dummy (i.e. zero) `return_modifier`, the same applies to parameters of primitive type. Their respective bytes just serve as placeholders.

3.3 Implementation with annotations

The second approach uses the new Java annotation facility. Our compiler automatically inserts annotations for fields, methods and classes. It is important to realize that we do not use annotations at the source code level, we only deal with them at the storage level, in the class file. That is, we insert them directly into the `RuntimeVisibleAnnotations` and `RuntimeVisibleParameter-Annotations` attributes, and not into the actual source code. The latter would duplicate information already available in the form of `peer`, `rep`, `readonly` and `pure` keywords. We will however use the textual form of annotations in examples to illustrate how classes, fields and methods are annotated, since showing the actual bytes would be rather cumbersome.

3.3.1 Annotation types

We use three different annotation types to store universe modifiers. All of them are annotated with the `Retention` annotation, with a value of `RetentionPolicy.RUNTIME`, meaning that they are accessible at run-time through reflective APIs.

UniverseClass

This annotation type serves the same purpose as the previously discussed `universe_class` attribute, namely it stores the version of the universe encoding used, and a boolean flag that indicates if the class has been compiled with run-time checks enabled.

```
@Retention(RetentionPolicy.RUNTIME)
public @interface UniverseClass {
    /** the version of the Universe type system used to compile the class */
    public String version();
    /** indicates if the class has run-time checks enabled or not */
    public boolean hasRuntimeSupport();
}
```

UniverseType

This annotation type is used to annotate fields, parameters and return types.

```
@Retention(RetentionPolicy.RUNTIME)
public @interface UniverseType {
    /** return the universe of the annotated type, encoded in a byte */
```

```
    public byte value();
}
```

Its only member, `value()`, is of type `byte` and contains the encoded universe modifier for the annotated field, parameter or return type, as described in 3.1. We will use this annotation to store the encoded universe type modifier for fields and method parameter and return types.

UniversePure

This annotation type has no members, these special annotations are called *marker annotations*. It indicates if the annotated method is pure. If the annotation is not present, the method is deemed non-pure.

```
@Retention(RetentionPolicy.RUNTIME)
public @interface UniversePure {
    /** empty marker annotation */
}
```

3.3.2 Classes

Classes are annotated with the `UniverseClass` annotation. A class that uses version 1.0 of the encoding, and which has run-time checks enabled, would be annotated as follows:

```
@UniverseClass(version="1.0", hasRuntimeSupport=true)
public class C {
    // ...
}
```

Both `version` and `hasRuntimeSupport` are given by the compiler, the latter is `true` if the compiler was invoked with the `dynchecks` universe option.

3.3.3 Fields

For fields of a reference type, an annotation of type `UniverseType` is generated. Its value is the encoded universe modifier. For example, the field `public rep` Object x; will become annotated with the `UniverseType` annotation as follows:

```
@universes.UniverseType(2) // => 10 binary == rep
public Object x;
```

3.3.4 Methods

For methods, we store purity with the `universes.UniversePure` annotation. If the method has a return value of reference type, we also add a `universes.UniverseType` annotation to the method, which holds the encoded universe modifier of the return type. If the method has parameters of reference type, we add a `universes.UniverseType` annotation to every such parameter. So a method with signature
`public pure readonly` Object m(`readonly` Object x) will become annotated as follows:

```
@UniversePure()
@UniverseType(3) // => 11 binary == readonly
public Object m(@UniverseType(3) Object x)
```

3.4 Class loading

When the compiler loads a class from a class file (as opposed to a source file), it does not know if universe information is present, and if it is, in which form (i.e. attributes or annotations).

3.4.1 Classes

For classes, there are three possible scenarios. Either a `universe_class` attribute is present, or a UniverseClass annotation, or neither. At present date the compiler first checks if the attribute is present, if it is, the compiler uses the information from the attribute. If the attribute is not present, it looks for the annotation. If that fails too, the compiler gives up, and assumes that the class has no universe information associated with it (i.e. no version string and no runtime-checks flag). Note that this means that if both an attribute and an annotation are present, the information in the attribute is used.

In the present implementation, absence of both the attribute and the annotation does not affect fields and methods. The compiler will try to find attributes or annotations on both fields and methods even if the class-wide information (e.g. the version of the encoding) is not present. This simplification was chosen because the classes representing class, field and method structures are modular, and introducing the dependency would have lead to much extra code. For instance if the compiler would look for universe information in a field structure, it would have to follow a link to the enclosing class structure, and first check if the class has a universe attribute or annotation.

3.4.2 Fields

For fields, the same three possibilities exist. Either a `universe_field` attribute is present, or a UniverseField annotation, or neither. Again the compiler first checks for the existence of the attribute and then looks for the annotation. If neither is present, the field (assuming it is of a reference type) is treated as being implicitly `peer`.

3.4.3 Methods

For methods, either the `universe_method` attribute is present, or a combination of annotations, or neither. The compiler first checks for existence of the `universe_method` attribute, and if it is present, gathers all the information from the attribute. If the `universe_method` attribute is not present, the compiler starts looking for annotations. If a UniversePure annotation is present, the method is considered pure, and non-pure otherwise. If a UniverseType annotation is present, the type modifier for the return type is read from it, otherwise the return type (again assuming it is a reference type) is treated as being implicitly `peer`. If the `RuntimeVisibleParameterAnnotations` attribute is present, the compiler then checks for every parameter if it is annotated with a UniverseType annotation, and sets the parameter type accordingly. If neither a `universe_method` attribute nor annotations are present, the method is treated as being non-pure, with both parameter and return types being implicitly `peer`.

3.5 Comparison of the two approaches

3.5.1 Reflection

While the annotation-based approach seems to be more complex than the simple attribute-based approach, it has one considerable advantage: Reflection is supported through the `java.lang.reflect` API. However, reflection will only work when using the most recent Java environment, J2SE 5.0. Earlier versions of the JVM are required to silently ignore the `RuntimeVisibleAnnotations` and its companion attributes. Also, the annotation types shown in 3.3.1 have to be distributed along with the program. Reflection can be made to work with the attribute-based

approach as well, by using a third-party bytecode processing libary like BCEL [3]. However, the resulting solution would be considerably less elegant.

3.5.2 Space usage

In terms of space, the attribute-based approach is more efficient. Every annotation takes 4 bytes of space for naming the annotation type, and giving the number of name/value pairs. Then for every name/value pair, 2 bytes are needed for identifying the name of the member, and an additional tag byte to identify the type of the value. Values are not stored ”inline”, but rather as items in the constant pool (e.g. integer constants), requiring an additional 2 bytes for the indirection.

3.5.3 Backward compatiblity

Another issue arises when generating class files with annotations. For reflection to work, the class file has to have its major and minor version set to 49.0 or higher[1]. Older Java virtual machines (e.g. the still widely used J2SE 1.4 VM) will then be unable to use this class. However, with rising acceptance of J2SE 5.0, we hope that this problem will eventually become insignificant.

3.5.4 Remarks

It is actually possible to use both implementations, but one does not gain much, since for annotations to be useful (i.e. making reflection possible) the class file version always has to be incremented. Also, this would definitely not be very space-efficient. Table 3.4 summarizes the advantages and disadvantages of the proposed implementations.

	Attributes	Annotations
Reflection	-	+
Space usage	+	-
Compatibility	+	-

Table 3.4: Comparison of the two approaches

[1] This is the value that the current version of `javac` included in J2SE 5.0 produces.

Chapter 4

Example

Let us now look at an example Java program with universe modifiers, and how these modifiers appear in the generated class file.

4.1 Example program

Listing 4.1 shows a simple stack program. The stack has the usual push/pop operations, along with a method to determine its current size. The stack is implemented internally as a linked list of nodes. The nodes form the representation of the stack, and therefore have the `rep` modifier. The stack stores references to arbitrary objects, which may be in any context, so they must be `readonly`. The size() method does not alter the state of the `Stack` object, so it has the `pure` modifier. The `Node` class is not shown here to save space.

```
public class Stack {
    public Stack() { top = null; }

    public void push(readonly Object data) {
        rep Node n = new rep Node(top, data);
        top = n;
    }

    public readonly Object pop() {
        readonly Object result = top.data;
        top = top.next;
        return result;
    }

    public pure int size() {
        int result = 0;
        readonly Node n = top;
        while (n != null) {
            n = n.next; result++;
        }
        return result;
    }

    private rep Node top;
}
```

Listing 4.1: A simple stack

4.2 Disassembling the example program

The Java disassembler `javap` allows us to disassemble the generated class file. Using the parameter `-verbose` it will also output the generated universe attributes. Since it does not know how to interpret them, it will just show the raw bytes. The bytecode of the methods is left out, for saving space, and because it contains no information related to this project.

4.2.1 Attributes

We now compile the `Stack` class by invoking the MultiJava compiler with the following command:
`java org.multijava.mjc.Main --universes Stack.java`
The `--universes` attribute enables parsing, typechecking and bytecode support using attributes. The universe attributes are underlined, and comments have been added to their data to help decipher it. The "polished" output of `javap` is shown below.

```
Compiled from "Stack.java"
public class Stack extends java.lang.Object
  org.multijava.universe_class: length = 0x3
   00 25 01 // encoding version is at constant_pool [37], run-time support enabled
  SourceFile: "Stack.java"
  Signature: length = 0x2
   00 17
  minor version: 3
  major version: 45
  Constant pool:
const #1 = Asciz      Stack;
const #2 = class      #1;
const #3 = Asciz      java/lang/Object;
const #4 = class      #3;
const #5 = Asciz      top;
const #6 = Asciz      LNode;;
const #7 = Asciz      Signature;
const #8 = Asciz      org.multijava.universe_field; // name of the universe field attribute
const #9 = Asciz      <init>;
const #10 = Asciz     ()V;
const #11 = Asciz     Code;
const #12 = NameAndType #9:#10;
const #13 = Method    #4.#12;
const #14 = NameAndType #5:#6;
const #15 = Field     #2.#14;
const #16 = Asciz     LineNumberTable;
const #17 = Asciz     org.multijava.universe_method; // name of the universe method attribute
const #18 = Asciz     pop;
const #19 = Asciz     ()Ljava/lang/Object;;
const #20 = Asciz     Node;
const #21 = class     #20;
const #22 = Asciz     data;
const #23 = Asciz     Ljava/lang/Object;;
const #24 = NameAndType #22:#23;
const #25 = Field     #21.#24;
const #26 = Asciz     next;
const #27 = NameAndType #26:#6;
const #28 = Field     #21.#27;
const #29 = Asciz     push;
const #30 = Asciz     (Ljava/lang/Object;)V;
const #31 = Asciz     (LNode;Ljava/lang/Object;)V;
const #32 = NameAndType #9:#31;
const #33 = Method    #21.#32;
```

```
const #34 = Asciz     size;
const #35 = Asciz     ()I;
const #36 = Asciz     org.multijava.universe_class; // name of the universe class attribute
const #37 = Asciz     1.0; // version of the universe encoding used
const #38 = Asciz     InnerClasses;
const #39 = Asciz     SourceFile;
const #40 = Asciz     Stack.java;

{
private Node top;
  Signature: length = 0x2
   00 06
  org.multijava.universe_field: length = 0x1
   02 // field "top" has rep modfier

public Stack();
  Signature: length = 0x2
   00 0A
  org.multijava.universe_method: length = 0x2
   00 00 // constructor is non-pure, and has no return type (special case)

public java.lang.Object pop();
  Signature: length = 0x2
   00 13
  org.multijava.universe_method: length = 0x2
   00 03 // pop() is non-pure, and its return type is readonly

public void push(java.lang.Object);
  Signature: length = 0x2
   00 1E
  org.multijava.universe_method: length = 0x3
   00 00 03 // push() is non-pure, its return type has no annotation, and its parameter is readonly

public int size();
  Signature: length = 0x2
   00 23
  org.multijava.universe_method: length = 0x2
   01 00 // size() is pure, and its return type has no annotation (primitive type)

}
```

Listing 4.2: javap output for attribute-based approach

Using the space-efficient attribute-based approach, the generated class file's size is 847 bytes.

4.2.2 Annotations

Now we use the annotation-based approach. We compile the class using the command:
`java org.multijava.mjc.Main --universesx=parse,check,annotations,dynchecks Stack.java`
Here we specify that we would like to use annotations instead of attributes. The `--universesx` option allows fine-grained control over the compiler's universe settings. Here we enable parsing, type checking, rum-time checks and output of annotations.

```
Compiled from "Stack.java"
public class Stack extends java.lang.Object
  RuntimeVisibleAnnotations: length = 0x10
   00 01 00 29 00 02 00 2A 73 00 2B 00 2C 5A 00 2D
   // This class has one annotation, its type is at pool[41] (UniverseClass)
   // The annotation has 2 members, the first member's name is at pool[42]
```

```
   // (version), and stores a string (tag = 0x73 = s), which lies at pool[43]
   // ("1.0"). The second member's name is at pool[44] (hasRuntimeSupport),
   // is of type boolean (tag = 0x5A = 'Z') and is represented by the
   // integer constant at pool[45] (1 = true)
  SourceFile: "Stack.java"
  Signature: length = 0x2
   00 19
  minor version: 0
  major version: 49
  Constant pool:
const #1 = Asciz      Stack;
const #2 = class      #1;
const #3 = Asciz      java/lang/Object;
const #4 = class      #3;
const #5 = Asciz      top;
const #6 = Asciz      LNode;;
const #7 = Asciz      Signature;
const #8 = Asciz      RuntimeVisibleAnnotations;
const #9 = Asciz      Lorg/multijava/universes/UniverseType;; // annotation type
const #10 = Asciz     value; // member of UniverseType annotation
const #11 = int 2;
const #12 = Asciz     <init>;
const #13 = Asciz     ()V;
const #14 = Asciz     Code;
const #15 = NameAndType #12:#13;
const #16 = Method    #4.#15;
const #17 = NameAndType #5:#6;
const #18 = Field     #2.#17;
const #19 = Asciz     LineNumberTable;
const #20 = Asciz     pop;
const #21 = Asciz     ()Ljava/lang/Object;;
const #22 = Asciz     Node;
const #23 = class     #22;
const #24 = Asciz     data;
const #25 = Asciz     Ljava/lang/Object;;
const #26 = NameAndType #24:#25;
const #27 = Field     #23.#26;
const #28 = Asciz     next;
const #29 = NameAndType #28:#6;
const #30 = Field     #23.#29;
const #31 = int 3;
const #32 = Asciz     push;
const #33 = Asciz     (Ljava/lang/Object;)V;
const #34 = Asciz     (LNode;Ljava/lang/Object;)V;
const #35 = NameAndType #12:#34;
const #36 = Method    #23.#35;
const #37 = Asciz     RuntimeVisibleParameterAnnotations;
const #38 = Asciz     size;
const #39 = Asciz     ()I;
const #40 = Asciz     Lorg/multijava/universes/UniversePure;; // annotation type
const #41 = Asciz     Lorg/multijava/universes/UniverseClass;; // annotation type
const #42 = Asciz     version; // member of UniverseClass annotation
const #43 = Asciz     1.0; // version of encoding
const #44 = Asciz     hasRuntimeSupport; // member of UniverseClass annotation
const #45 = int 1;
const #46 = Asciz     InnerClasses;
const #47 = Asciz     SourceFile;
const #48 = Asciz     Stack.java;
```

```
{
private Node top;
  Signature: length = 0x2
   00 06
  RuntimeVisibleAnnotations: length = 0xB
   00 01 00 09 00 01 00 0A 42 00 0B
   // The field top has one annotation, its type is at pool[9] (UniverseType).
   // The annotation has one member, its name is at pool[10] (value), and
   // its value is a byte (tag = 0x42 = 'B') stored in the integer constant
   // at pool[11] (2 = 10 binary = rep).

public Stack();
  Signature: length = 0x2
   00 0D

public java.lang.Object pop();
  Signature: length = 0x2
   00 15
  RuntimeVisibleAnnotations: length = 0xB
   00 01 00 09 00 01 00 0A 42 00 1F
   // The method pop() has one annotation, its type is at pool[9] (UniverseType).
   // The annotation has one member, its name is at pool[10] (value), and
   // its value is a byte (tag = 0x42 = 'B') stored in the integer constant
   // at pool[31] (3 = 11 binary = readonly). This means that the return type
   // of the method has the readonly modifier.

public void push(java.lang.Object);
  Signature: length = 0x2
   00 21
  RuntimeVisibleParameterAnnotations: length = 0xC
   01 00 01 00 09 00 01 00 0A 42 00 1F
   // The method push() has one parameter, that parameter has one annotation,
   // its type is at pool[9] (UniverseType). The annotation has one member
   // named "value" of type byte, represented by the integer constant at
   // pool[31] (3 = 11 binary = readonly). This means the first parameter
   // of the method has the readonly modifier.

public int size();
  Signature: length = 0x2
   00 27
  RuntimeVisibleAnnotations: length = 0x6
   00 01 00 28 00 00
   // The method size() has one annotation, its type is at pool[40] (UniversePure)
   // The annotation has no members (i.e. it is a marker annotation).

}
```

Listing 4.3: javap output for annotation-based approach

The size of the class file generated by the annotation-based approach is 988 bytes, making it a total of 141 bytes larger than the one generated with the attribute-based approach.

Chapter 5

MultiJava compiler and JML extensions

Several extensions were made to the MultiJava [2] compiler `mjc` and related packages, only a high-level overview is given here.

5.1 Attributes

5.1.1 New attribute classes

Several new classes were introduced for supporting the universe attributes defined in the previous chapters, and the attributes used for annotations (the MultiJava compiler does not yet support annotations).

Package org.multijava.util.classfile

- `UniverseFieldAttribute`: Implements a universe field attribute.
- `UniverseMethodAttribute`: Implements a universe method attribute.
- `UniverseClassAttribute`: Implements the universe class attribute.

All attributes inherit from `org.multijava.util.classfile.Attribute`. The universe attributes are just wrappers for their uninterpreted, raw data values (e.g. a single byte for `UniverseFieldAttribute`). Two wrapper classes, `CUniverseTypeAnnotation` and `CUniverseMethodAnnotation` were added to the package `org.multijava.mjc` to simplify code. They basically parse the raw data provided by the universe attributes and provide accessor functions.

5.1.2 Attribute parsing

The MultiJava compiler (`mjc`) uses a *chain of responsibility* pattern (described in [6]) to parse attributes from class files. The attribute is passed along a chain of attribute parsers, until some parser recognizes it, or it reaches the end of the chain, in which case the attribute cannot be recognized. So we simply create a new class `UniverseAttributeParser` which implements an abstract `AttributeParser`, and append the singleton instance to the chain. This parser recognizes the universe attributes, and returns objects of type `UniverseFieldAttribute`, `UniverseMethodAttribute`, and `UniverseClassAttribute`.

To support annotations, the existing attribute parser `BaseAttributeParser` had to be extended as well, enabling it to recognize the `RuntimeVisibleAnnotations` attribute and its companion attributes.

All these classes are located in the package `org.multijava.util.classfile`.

5.2 Annotations

Apart from the new annotation attributes, several annotation-related classes were added to the `org.multijava.util.classfile` package as well. These represent the data structures used for storing annotations in the class file, as well as the attributes that are added to annotated program elements.

Package org.multijava.util.classfile

- `Annotation`: Represents the `annotation` data structure. Holds a list of element/value pairs.
- `Annotation.ElementValuePair`: Nested class representing a name/value pair stored in an `annotation` structure. Contains an `ElementValue`.
- `ElementValue`: Abstract class representing a value in a name/value pair.
- `AnnotBooleanElementValue`, `AnnotByteElementValue` , ...: Concrete classes representing different types of values in `annotation` structures.
- `RuntimeVisibleAnnotationsAttribute`: Implements a RuntimeVisibleAnnotations attribute as specified in [10].
- `RuntimeVisibleParameterAnnotationsAttribute`: Implements a RuntimeVisibleParameter-Annotations attribute as specified in [10].

These classes represent the low-level infrastructure for the annotations facility, and are implemented in a reusable way. Full support for annotations (on the source code level) can be built on top of these classes.

The universe annotation types described in 3.3.1 were added to the `org.multijava.universes` package.

Package org.multijava.universes

- `UniverseClass`: Annotation type for storing the version of the encoding and availability of run-time checks.
- `UniverseType`: Annotation type storing an encoded type modifier.
- `UniversePure`: Marker annotation type for method purity.

5.3 Disassembler

Some straightforward modifications to the disassembler `org.multijava.dis` were made, namely to the `Disassembler` class, to support displaying of universe information from `.class` files. This is useful because it allows us to create testcases where we compare the output of the disassembler to the expected output and then check if everything has been written to the class file correctly. The modifications were minimal, since most of the work is already done in other classes. Again we insert our `UniverseAttributeParser` instance into the chain of responsiblity to gain access to the extended information stored in the universe attributes.

Package org.multijava.dis

- `Main`: Insert universe attribute parser into chain of responsibility.
- `Disassembler`: Changed output functions to also output universe related information.
- `DisOptions`: Added new command-line switch `-e (--universes)` to toggle output of universe information.

5.4 Modifications to compiler

Several modifications to the compiler `org.multijava.mjc` had to be made in order to intergate the new functionality.

Package org.multijava.mjc

- `Main`: Insert universe attribute parser into chain of responsibility
- `CType`: Extended signature parsing methods to call getTypeRep() with correct universe modifier
- `CClassType, CArrayType`: New method to get the type's encoded universe modifier
- `CUniverse, CUniversePeer, CUniverseImplicitPeer, CUniverseRep, CUniverseReadonly`: new methods to get the byte constant representing a particular universe
- `CSourceMethod`: Add universe method attribute or annotation to MethodInfo output
- `CBinaryMethod`: Pass universe modifiers to type loader
- `CField`: Add universe field attribute or annotation to FieldInfo output
- `CBinaryField`: Pass universe modifier to type loader
- `CClass`: Add method for determining Universe version and availability of run-time checks
- `CSourceClass`: Add universe class attribute or annotation to ClassInfo output
- `CBinaryClass`: Enable access to version of universe encoding and availability of run-time checks

+ `CUniverseTypeAnnotation, CUniverseMethodAnnotation`: new wrapper classes for universe fields and methods

5.4.1 Modifications to class file utility classes

Several classes in `org.multijava.util.classfile` had to be extended to enable convenient access to universe information stored in attributes or annotations.

Package org.multijava.util.classfile

- `ClassInfo`: New method to extract the version of the Universe type system used to compile this class, and check if the class has run-time checks enabled.
- `FieldInfo`: New method to extract the universe modifier from a universe field attribute or annotation.
- `MethodInfo`: New methods to extract purity and universe modifiers for return and parameter types from a universe method attribute or annotations.

+ `UniverseByteConstants`: New class containing utility methods to work with the byte constants representing universe modifiers in annotations and attributes.

5.5 Testcases

A number of very simple testcases were created in the `org.multijava.mjc.testcase.universes.-codegen` package to test if the universe modifiers are written correctly to the class file. Separate testcases are provided for both the attributes-based and the annotation-based approach. They work by first compiling a number of universe-enabled Java source files, then disassembling the resulting class files and comparing the output of the disassembler with the expected output. This is one of the reasons that the disassembler was modified, so it could be used to validate testcases, otherwise the disassembler would ignore universe attributes or annotations. The testcases are integrated into the MultiJava testing framework, and can be invoked by executing the `runtests` target of the project's Makefile.

5.6 JML

The Java Modelling Language [8] is a behavioral interface specification language that can be used to specify the behavior of Java classes. The JML tools are built on top of the MultiJava compiler and its utility classes, so they can immediately benefit from the extensions described in the previous chapters.

5.6.1 jmlspec

The `jmlspec` (`org.jmlspecs.jmlspec`) tool generates specification skeletons from Java source files, and also from class files. In the latter case it can now also extract universe information for fields and methods when invoked with the `--universes` command-line switch. The only class that had to be slightly modified is `JspBinaryPrinter`, changes being minimal. The modifications were similar to the ones in the disassembler (see 5.3), since all the actual work of reading universe annotations and attributes is done in the MultiJava class file packages.

The specification skeleton generated by `jmlspec` when invoked with the class file for the `Stack` example is shown in listing 5.1. Without the changes, the universe modifiers would not have been extracted from the class file.

```
// Generated by jmlspec from the .class file


public class Stack extends java.lang.Object {

    // CLASS SPECIFICATIONS
    /*@
      @
      @*/

    // FIELDS

    // METHODS AND CONSTRUCTORS

    /*@
      @
      @*/
    public Stack();

    /*@
      @
      @*/
    public /*+@ readonly @+*/ java.lang.Object pop();
```

```
    /*@
      @
      @*/
    public void push(/*+@ readonly @+*/ java.lang.Object Param0);

    /*@
      @
      @*/
    public /*+@ pure @+*/ int size();
}
```

Listing 5.1: jmlspec output for `Stack.class`

Chapter 6

Conclusion and future work

6.1 Conclusion

Bytecode support for the Universe type system can be implemented in a way that does not conflict with the Java class file format. Two solutions to the problem have been implemented. With rising adoption of J2SE 5.0, it is expected that the annotation-based approach will prevail, because of its favorable properties that allow reflection through the standard API. However, the attribute-based approach is less demanding in terms of space, and does not have any backward compatibility issues. It is therefore the preferred solution for the time being.

6.2 Bytecode verification

When the JVM attempts to load a class from a `.class` file, it first checks for basic integrity of the class file, i.e. correct magic number, proper length of attributes, no superficial information in constant pool, and so on. Then the verifier proceeds and performs some additional verification, including checks if every class (except `Object`) has a direct superclass, `final` classes are not subclassed or `final` methods overridden and so on.

When these static constraints are fulfilled, the verifier checks the bytecode for every method by performing a data-flow analysis using an *abstract interpreter*. The abstract interpreter models the effect of instructions on the operand stack and local variables at a type level. For instance, if the instruction uses a local variable, it checks if the specified variable contains a value of the correct type. If it uses a value on the operand stack, it checks if the operand stack has a value of the correct type on top. The exact process is described in detail in [9].

For the Universe type system, an extended verifier which takes the universe types into consideration is needed. In particular, it needs to check if the smallest common super type on the stack or in a register is compatible with the instruction (e.g. no invocations of non-pure methods on readonly references). The extended verifier could use the information stored in attributes or annotations to gain the needed type information, so no additional information has to be stored.

6.3 Reflection

Both solutions allow reflection, though the annotation-based approach is preferred because annotations are accessible directly through the J2SE 5.0 Reflection API, so no additional libraries are needed. The information gained through reflection could be used by software development tools for a variety of purposes. Reflection is also useful for the optimization of runtime support, as described below.

6.4 Optimization of Runtime Support

Runtime support for the Universe type system, as described in [12], can be optimized when it is known if a compiled class has runtime support enabled. In particular, object creation can be facilitated. When runtime support is enabled, newly created objects are registered in an *owner hashtable*, to store the ownership relation between objects. A problem arises when the newly allocated object attempts to create new objects itself in its constructor, because those objects have to be registered in the hashtable relative to the current object, which at that point in time has not been registered yet, because the registration happens after the constructor is called. Therefore, a second registration of the object is done at the beginning of the constructor, before any other objects are created. If it is known beforehand that the class is universe-aware, the call to the hashtable method after the constructor is not necessary, because classes with runtime support register themselves when instantiated.

Unfortunately this optimization can be problematic because the class of the instantiated object could later be recompiled without runtime support. This would lead to the object no longer being entered into the owner hashtable, because the code that creates the object was not recompiled as well, and therefore still assumes that the class is universe-aware.

Bibliography

[1] B. Bokowski and J. Vitek. Confined types. In *Proceedings of Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, ACM SIGPLAN Notices, 1999.

[2] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. Multijava: Design rationale, compiler implementation, and user experience. Technical Report 04-01, Iowa State University, Dept. of Computer Science, January 2004. Submitted for publication.

[3] Markus Dahm. BCEL: The Bytecode Engineering Library. http://jakarta.apache.org/bcel/.

[4] IBM DeveloperWorks. Annotations in Tiger, Part 1: Add metadata to Java code. http://www-106.ibm.com/developerworks/java/library/j-annotate1/.

[5] IBM DeveloperWorks. Annotations in Tiger, Part 2: Custom annotations. http://www-106.ibm.com/developerworks/library/j-annotate2.html.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[7] Sun Microsystems Inc. JSR 175: A Metadata Facility for the Java Programming Language. http://www.jcp.org/en/jsr/detail?id=175.

[8] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06x, Iowa State University, Department of Computer Science, 2004. See www.jmlspecs.org.

[9] X. Leroy. Java bytecode verification: algorithms and formalizations. *Journal of Automated Reasoning*, 30(3–4):235–269, 2003.

[10] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., 1999.

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

[12] Daniel Schregenberger. Runtime checks for the Universe type system. Semester Thesis, October 2004.