Runtime checks for the Universe type system

Daniel Schregenberger

Semester Project Report

Software Component Technology Group
Department of Computer Science
ETH Zurich

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

12th June 2005
(originally submitted 7th October 2004)

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

Abstract

The goal of this semester project is to design and implement runtime checks for the Universe type system.

This involves the design of a method to store the ownership relation at runtime, the implementation in the Universe Compiler, the generation of the bytecode for the runtime tests and the creation of testcases for the dynamic checks.

Chapter 1 explains the motivation for this semester project by giving a short introduction to the Universe type system and then discussing the runtime tests in Java and Universe Java.

Possible approaches for data structures are discussed in chapter 2.

Chapter 3 provides a description of the runtime tests in terms of sourcecode transformations.

The actual implementation is then described in chapter 4.

Finally chapter 4.4 describes the test cases that have been developed to ensure the functionality of the dynamic checks and in chapter 5 the solution is discussed and analyzed.

Contents

1 Motivation **4**
1.1 The Universe type system 4
1.1.1 The extended types of the Universe type system 4
1.2 Runtime checks in Java 5
1.3 Runtime checks in the Universe type system 5
1.3.1 Method invocation 5
1.3.2 The `instanceof` operator and casts of reference types 6
1.3.3 Assignment to an array component of reference type 7
1.3.4 Exception handling 7

2 Different approaches for a solution **8**
2.1 Modification of the virtual machine 8
2.2 Modifying `java.lang.Object` 8
2.2.1 Compiling an own `java.lang.Object` classfile 8
2.2.2 Bytecode manipulation 9
2.3 Global Hashtable 9
2.4 Summary 10

3 Sourcecode transformations **11**
3.1 Objects 11
3.1.1 New objects 11
3.1.2 The `instanceof` operator 13
3.1.3 Casts 14
3.2 Arrays of reference types 14
3.2.1 Creation of new arrays 14
3.2.2 The `instanceof` operator 15
3.2.3 Casts 15
3.2.4 Assignments to array elements 16

4 Implementation **17**
4.1 The owner Hashtable 17
4.1.1 The runtime classes 18
4.1.1.1 The policy classes 19
4.2 Various aspects of this solution 19
4.2.1 Initialization 19

4.2.2 Field initialization and object creation in constructors 20
4.2.3 Static functions 21
4.2.4 Pure methods 22
4.2.5 Legacy code 22
4.3 Usage 22
4.3.1 Compiling and running Universe programs with runtime check support . . . 22
4.3.2 Using alternative implementations of the runtime classes 22
4.3.2.1 Implementation classes 23
4.3.2.2 Policy classes 23
4.3.3 Writing an own runtime class 24
4.3.4 Distributing programs that use Universe runtime checks 24
4.4 Testcases 24
4.5 The changes to the MultiJava Compiler 25

5 Evaluation 27
5.1 Benchmarks 27
5.1.1 Objecttest 27
5.1.2 Stringtest 27
5.1.3 Listtest 28
5.1.4 Summary 28
5.2 Code comparison 28
5.3 Conclusion and open issues 29

Bibliography 31

A Sun Licence 32

B Changes since the initial submission 33

C Sourcecode 34
C.1 Runtime Classes 34
C.1.1 Initialization 34
C.1.2 Interfaces 37
C.1.3 Implementations 42
C.1.4 Policies 64
C.2 Benchmarks 71
C.3 Code comparison 76

Chapter 1

Motivation

1.1 The Universe type system

The Universe type system, as described in [MPH01], is a type system designed to control aliasing and dependencies. It works by hierarchically partitioning the object store into so-called universes.

Every object is assigned an owner. A universe then is a set of objects that share the same owner. A special case is the root universe, which holds the objects that do not have an owner. For example objects created in the static main function cannot have an owner, simply because there is none. Normal read-write references are allowed only between objects in the same universe or from the owner to objects in its universe. Any other reference is read-only. So if an object X has a reference to object Y, either X is the owner of Y, X and Y are in the same universe or the reference is read-only.

The idea of read-only references needs some explanation. Normal references in Java are read-write and allow any modification of the referenced object. The idea of read-only references is that they do not allow such modifications. Neither by accessing fields directly nor by calling functions of that object that modify fields[1]. Furthermore read-only references are transitive so you cannot get a read-write reference through a read-only reference.

This alias control enforces representation encapsulation and avoids leaking, while, in combination with the concept of read-only references, it does not restrict aliasing too much and does therefore not enforce the very expensive alternative of cloning objects.

1.1.1 The extended types of the Universe type system

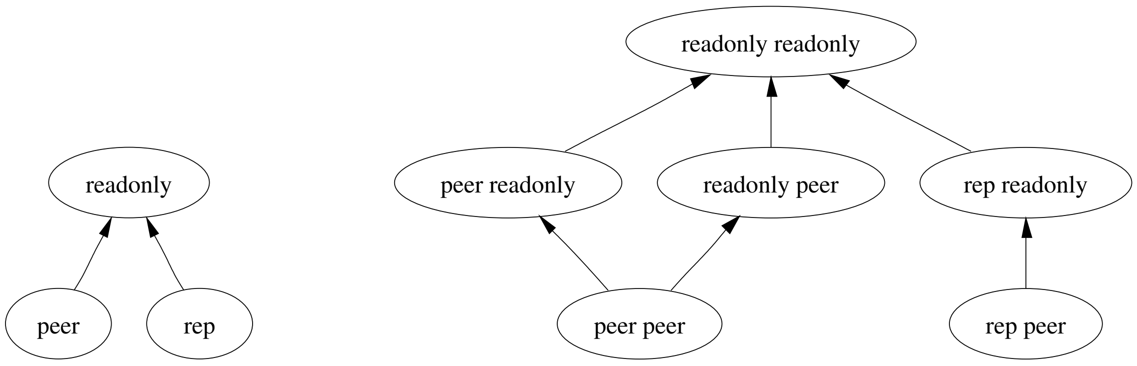
The classification of references is done by using an extended type system. Normal references are of type `peer`, references of an object A to an object in the universe of A are of type `rep` and read-only references, pointing anywhere, are of type `readonly`. The relation between these three types is shown in figure 1.1(a).

By the substitution principle follows that whenever a readonly reference is used, it might point to a `peer` or `rep` object. `readonly` objects do not exist. Think of `readonly` the same way you think of interfaces or abstract base classes. References of that type are allowed, but you cannot create actual instances.

`readonly`, `peer` and `rep` are attributes of the reference, not 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 basic types behave exactly the same way, but arrays of reference types are slightly more complicated, because not only the array object has an owner, but also the elements are of an

[1] An exception are side effect free functions, called `pure` in the Universe type system.



(a) for objects and arrays of basic types

(b) for arrays of reference typ

Figure 1.1: Type hierarchy graphs for the Universe type system

extended type. Figure 1.1(b) shows the Universe type hierarchy for arrays. See [DMS05, DM05] for more information about the handling of arrays in the Universe type system.

1.2 Runtime checks in Java

Object-oriented languages owe a big part of their dynamics to polymorphism and the substitution principle. This kind of dynamics makes it impossible to statically type check everything. Some languages, like for example C++, do not do many dynamic checks and accept the risk of nasty crashes due to segmentation faults or other violations of the operating system rules.

In Java there are five places, where the actual class of a referenced object affects program execution in a manner, that cannot be deduced from the static type of the expression:

1. Method invocation,
2. the `instanceof` operator,
3. casts of reference types,
4. assignment to an array component of reference type and
5. exception handling.

1.3 Runtime checks in the Universe type system

The extended types of the Universe type system have no influence on the dynamic checks needed for method invocation and exception handling, so we only need to adapt the others to the new requirements. The next few sub chapters provide an overview over the five runtime checks in Universe Java.

1.3.1 Method invocation

Method invocation does not need to be handled, because the different types of the Universe type system do not introduce different methods and therefore do not influence dynamic method binding.

1.3.2 The instanceof operator and casts of reference types

Listing 4.7 shows an example why we need runtime checks for casts and the `instanceof` operator. Figure 1.2 illustrates this situation. The scenario is a doubly linked list with iterators. If we want to change some list element using an iterator, we have to pass the iterators `readonly` reference to the list head. The list head will then try to cast the `readonly` reference to a `rep` reference and, if that succeeded, perform the modification we requested.

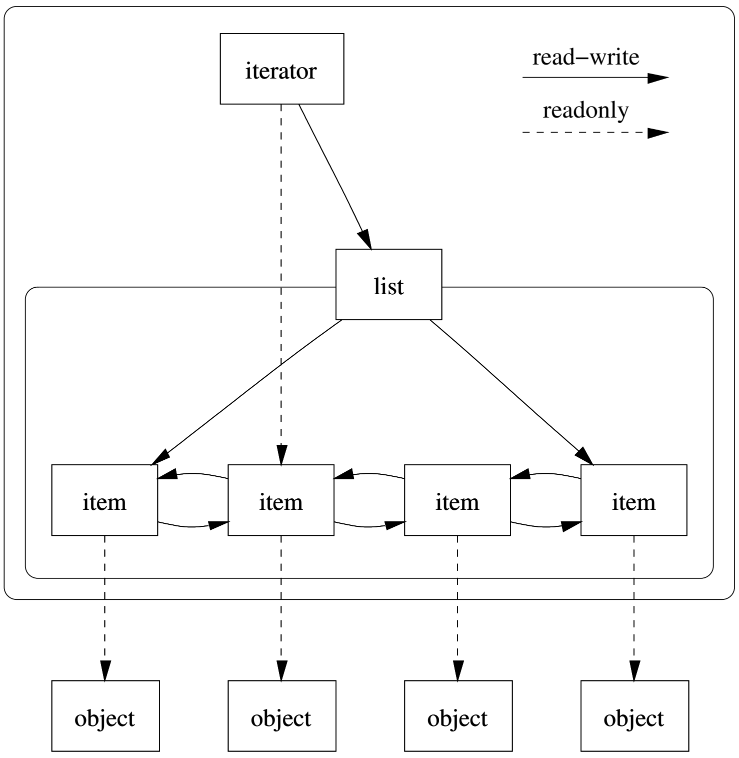


Figure 1.2: A doubly linked list with an iterator.

```
class Item {
    int key;
    readonly Object data;

    peer Item next;
    peer Item prev;
}

class List {
    rep Item head;
    rep Item tail;

    public List () {
        // ...
    }

    public void update (readonly Item i, readonly Object data) {
        if ( i instanceof rep Item ) {
            rep Item j = (rep Item) i;
            j.data = data;
        } else
            throw new Exception("The specified Item does not belong to this List.");
    }
}

class Iterator {
    readonly Item current;
    peer List l;

    public Iterator (List l) {
        // ...
    }

    public readonly Item next () {
        // ...
    }
}

class Example {
    public void dostuff {
        peer List l = new List();
        // ...
        peer Iterator iter = new Iterator(List);
        readonly Item i;
        while ( (i = iter.next()) != null )
            l.update(i, new Object());
    }
}
```

Listing 1.1: Example for the use of the instanceof operator and casts in Universe Java

1.3.3 Assignment to an array component of reference type

The Java Language Specification [Suna] states[2] that type-checking rules allow the array type `T[]` to be treated as a subtype of `S[]` if `T` is a subtype of `S`, but this requires a run-time check for assignment to an array component, similar to the check performed for a cast.

Applied to our extended type system, this applies to array references declared to hold `readonly` elements: they might actually reference an array of `peer` elements. Therefore before any assignment to an array component, we must assure that we are not writing a `readonly` or `rep` object into an array of `peer` elements.

Listing 1.2 shows an example for an illegal array store expression, that is statically correct but should generate an error at runtime.

```
peer readonly Object[] a = new peer peer Object[5];
a[0] = new rep Object();
```

Listing 1.2: Example for an illegal array store expression

1.3.4 Exception handling

Exception handling does not change, since the Universe type system uses `readonly` exceptions as described in chapter 2.4 of [DM04]. On creation exceptions are normally of type `peer` and when catched, they can be handled as `readonly`.

[2]in chapter 10.10: Array Store Exception

Chapter 2

Different approaches for a solution

2.1 Modification of the virtual machine

A straightforward approach would be to modify the Java virtual machine. Straightforward since the tests are done at runtime and the standard tests described in the Java Language Specification [Suna] are also implemented in the VM.

This approach would involve dynamic runtime data structures, to store the owner for every object created in the program, and tests operating on these data structures. The data structures could be implemented in the VM code and would only require the class files to contain information about the extended types.

But one of the main goals of this project was to avoid any modification of the virtual machine and therefore be able to run programs with Universe runtime checks on the standard JVM delivered by Sun Microsystems. The idea behind this restriction is the simple thought that (almost) everyone using Java has the VM by Sun or a compatible one. Modifying the VM would require people interested in trying or using the Universe type system to install this modified VM. While this would certainly not be a big issue for developers, it would also affect anyone wanting to run a Universe Java program with runtime checks.

2.2 Modifying `java.lang.Object`

The next idea was to provide the additional attributes required for the Universe model by providing an alternative class file of the `java.lang.Object` class. Since every class inherits from `java.lang.Object`, this would enhance all objects, no matter whether they are in the standard classpath or user written.

The actual tests would then consist of sourcecode transformations, either realized by writing a preprocessor for Universe Java code or by implementing them directly into the compiler. These transformations would then store any data needed for the tests in the additional attributes and also get it again from there for the tests.

There are two possible ways to implement this approach, a short discussion follows.

2.2.1 Compiling an own `java.lang.Object` classfile

The GNU Classpath project [GNU] is creating free (in the sense of free speech) core libraries to be used with, hopefully also free, Java virtual machines.

This approach looked promising, so I tried to add an attribute to `java.lang.Object` to make a proof of concept before starting with the real work.

Modifying the source code was easy, since it is normal Java code. But running applications using the modified `java.lang.Object` classfile unfortunately was not that easy. When trying to run a simple hello world program with only Object.class modified but all other core classes being the original ones from the JDK, the following error occurred:

```
# java -Xbootclasspath/p:/home/scdaniel/lib testclass
Error occurred during initialization of VM
java.lang.IllegalStateException
```

This indicates that the GNU classpath project and the JDK are not, or not yet, 100% bytecode compatible.

Compiling all core classes and replacing the JDK by the GNU classpath did not work either, because some classes are still missing in the GNU classpath project[1]:

```
# java -Xbootclasspath/p:/home/scdaniel/lib testclass
Error occurred during initialization of VM
java.lang.NoSuchMethodError:
java.lang.ClassLoader.findNative(Ljava/lang/ClassLoader;Ljava/lang/String;)J
```

2.2.2 Bytecode manipulation

An alternative method to provide modified class files is the use of bytecode engineering utilities, such as BCEL [BCE], to directly modify the classfile.

Unfortunately most of these tools are too clever and assume the superclass is `java.lang.Object`, if the user specified none. But the original `java.lang.Object` classfile uses `null` as its superclass, since it is the root of the class hierarchy. Therefore using these tools to do our modification did not work, because it created a cyclic dependency. However, using a slightly customized version of Jasmin [Mey][2], I created for this task, did not work either and resulted in a crash of the virtual machine during start.

Adding methods worked but adding attributes did not. This indicates that `java.lang.Object` is coupled very tight to the virtual machine. The GNU classpath project provides a virtual machine integration guide [Rea] and states therein, that VM developers need to implement some of the classes in the VM. While `java.lang.Object` is not mentioned explicitly there, it seems it is nevertheless heavily depending on such VM hooks.

Apart from all that, section D of the supplemental licence terms of the Binary Code Licence Agreement for the JDK[3] forbids any changes to class files in the java namespace as well as the creation of additional class files in this namespace.

2.3 Global Hashtable

Another approach would be to register every newly created object in a hashtable and look up owners in the hashtable for the checks.

The advantage of this solution is that it is entirely realizable with standard Java. Either by using a preprocessor or by generating the transformations in the compiler. A disadvantage is that the class providing access to the hashtable through static functions needs to be distributed with Universe Java programs, but I think this is only a minor issue.

[1] Version 1.0 of the GNU Classpath project is intended to be fully compatible with the 1.1 and largely compliant with the 1.2 API specification and will have a stable API for interacting with virtual machines.
The version used for the evaluation of this approach was the current version from CVS of April 2004: 0.08.

[2] Jasmin is a Java assembler interface that translates an assembler like language into Java bytecode.
There are various tools that decompile Java class files to Jasmin code. I have used `JasminVisitor.java` which is part of BCEL [BCE].

[3] http://java.sun.com/j2se/1.4.2/j2sdk-1_4_2_05-license.txt
See section A in the appendix for a copy of that licence section.

2.4 Summary

Modifying the VM should be avoided for this project, if possible. Manipulating the original `java.lang.Object` does not work and is forbidden. Compiling a customized `java.lang.Object` classfile might be a good solution in the future when the GNU Classpath project reaches a level that makes it a drop-in replacement to the JDK by Sun. This leaves us with the Global Hashtable as the only acceptable solution right now.

Before the actual implementation is described in chapter 4, in the next chapter the sourcecode transformations are presented.

Chapter 3

Sourcecode transformations

In this chapter I will describe the sourcecode transformations needed for the runtime checks. To make myself clear I need to define a few conventions first:

- `T` denotes a type.
- `S` denotes a supertype of `T` but by the substitution principle follows that wherever `S` is used also `T` could be used.
- Variables are called `x` and `y`.
- Temporary variables, created by the transformation only for the checks, are named `tmp`.

As described in chapter 2 we need to save the owner of every object. For simplicity in terms of readability and to not be dependent on the current implementation, I have decided to use a simple notation that looks like the approach described in chapter 2.2:

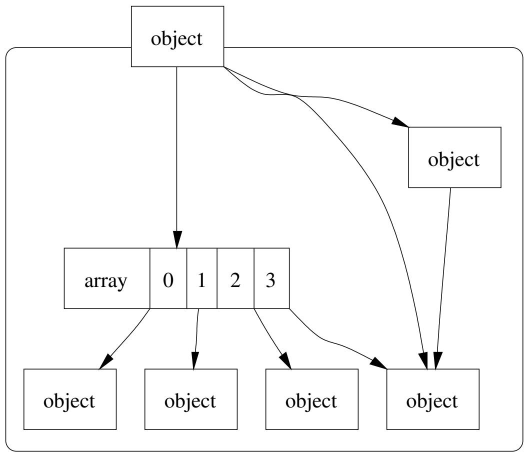
- Every object has a field named `$owner` that is used to store a reference to the objects owner.
- While arrays of basic types can be handled the same as normal objects, arrays of reference types need to be treated in a special way, since they not only have an owner themselves, but also their elements are of an extended type. Thus we need an additional field to store the owner type of the elements.
  So for arrays of reference types, every array object has a field named `$type` that stores the Universe type of the elements it holds.
  This second field is a little more complicated and needs some explanation. It is always relative to the array object and by definition only `peer` and `readonly` are allowed as modifiers. If `rep` would be allowed, the array object itself would be the owner of the entries in the array and therefore only `readonly` references to elements would be allowed from the outside. It would even be impossible to create such objects since an array is never owner of a context.
  Figure 3.1 shows the situation for different element types.

So much for the simple notation used for the data we need to store with every object. The data structure used in the actual implementation is described in chapter 4.

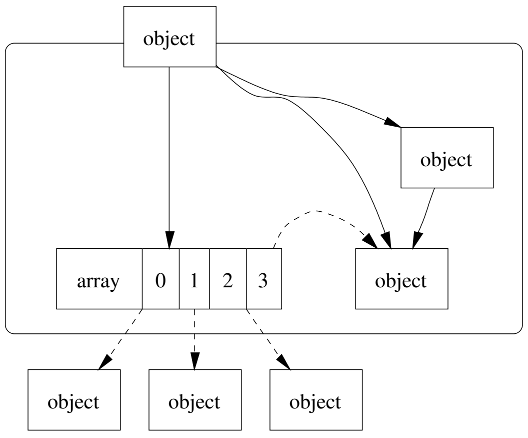
3.1 Objects

3.1.1 New objects

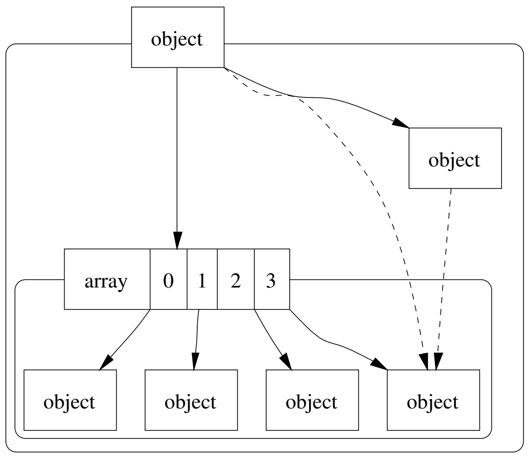
For every newly created object, including anonymous ones, we need to set its owner.



(a) element type `peer`



(b) element type `readonly`



(c) element type `rep` (if it was allowed)

Figure 3.1: Illustration of the element type for arrays.

New objects are either of type `peer` or `rep`. There is no such thing as readonly objects. `readonly` is an attribute of the reference and not of the object.

For new `peer` objects, the owner is the same as for the current object:

```
new peer T();
```

Listing 3.1: `new peer` before the transformation.

```
T tmp = new T();
tmp.$owner = this.$owner;
```

Listing 3.2: `new peer` after the transformation.

For new `rep` objects, the owner is the current object:

```
new rep T();
```

Listing 3.3: `new rep` before the transformation.

```
T tmp = new T();
tmp.$owner = this;
```

Listing 3.4: `new rep` after the transformation.

3.1.2 The `instanceof` operator

Extending the type system also involves updating the `instanceof` operator, to be able to check the dynamic type of a variable at runtime.

This of course only makes sense, when testing a `readonly` variable whether it is actually containing a `rep` or `peer` variable. If the variable is not `readonly`, the compiler can statically determine if the statement is valid with regards to Universe types and then either report an error or omit the Universe specific additional test. This for example happens if a `rep` variable is checked if it contains an object of a certain class:

```
rep S x;
// ...
if ( x instanceof rep T )
    // ...
```

Listing 3.5: An example for using `instanceof` on a `rep` variable.

In this case, the additional Universe test can be omitted and it therefore is equivalent to a normal `instanceof` test.

The additional checks can be done by comparing the owner field of the checked object and the current object, referenced by `this`.

```
(x instanceof peer T)
```

Listing 3.6: `instanceof peer` before the transformation.

```
((x instanceof T) &&
    (x.$owner == this.$owner))
```

Listing 3.7: `instanceof peer` after the transformation.

```
(x instanceof rep T)
```

Listing 3.8: `instanceof rep` before the transformation.

```
((x instanceof T) &&
    (x.$owner == this))
```

Listing 3.9: `instanceof rep` after the transformation.

The `instanceof readonly` check is a special case. `instanceof` evaluates to true whenever the object, that is checked, is of the specified type or a subtype of it. By definition, `readonly` is supertype of `rep` and `peer` and therefore any object can be handled as if it was `readonly`, thus allowing us to omit the additional Universe runtime check for this special case.

```
(x instanceof readonly T)
```

Listing 3.10: `instanceof readonly` before the transformation.

```
(x instanceof T)
```

Listing 3.11: `instanceof readonly` after the transformation.

3.1.3 Casts

Downcasts require dynamic checking, whereas upcasts are always permitted. Casting a `rep` object to a `peer` object or vice versa is illegal by definition and will be detected by the compiler.

```
readonly S x;
peer T y = (peer T) x;
```

Listing 3.12: A cast to `peer` before the transformation.

```
S x;
if ( x.$owner != this.$owner )
    throw new ClassCastException();
T y = (T) x;
```

Listing 3.13: A cast to `peer` after the transformation.

```
readonly S x;
rep T y = (rep T) x;
```

Listing 3.14: A cast to `rep` before the transformation.

```
S x;
if ( x.$owner != this )
    throw new ClassCastException();
T y = (T) x;
```

Listing 3.15: A cast to `rep` after the transformation.

3.2 Arrays of reference types

3.2.1 Creation of new arrays

In addition to the owner of the array, we need to save the type of the elements in the array. So the sourcecode transformation for the creation of new arrays is as follows.

```
new peer <type> T[10];
```

Listing 3.16: `new peer` array before the transformation.

```
T[] tmp = new T[10];
tmp.$owner = this.$owner;
tmp.$type = <type>;
```

Listing 3.17: `new peer` array after the transformation.

```
new rep <type> T[10];
```

Listing 3.18: `new rep` array before the transformation.

```
T[] tmp = new T[10];
tmp.$owner = this;
tmp.$type = <type>;
```

Listing 3.19: `new rep` array after the transformation.

Where `<type>` may be either `peer` or `readonly`. `rep` is not allowed, since the element type is always relative to the type of the array. If `rep` was allowed, the array would be the owner of the elements it contains, which would restrict any access from outside to `readonly`, which is not desirable.

3.2.2 The `instanceof` operator

The check for the dynamic type of array references also depends on the type of the elements, therefore we also need to check the element type for compliance.

For the following transformations `<type>` may be `peer` or `readonly`. If it is `readonly`, the check for the element type is omitted, to provide the same behaviour as the standard Java `instanceof` operator.

```
(x instanceof peer <type> T[])
```

Listing 3.20: `instanceof peer` array before the transformation.

```
( (x instanceof T[]) &&
    (x.$owner == this.$owner) &&
    (x.$type == <type>) )
```

Listing 3.21: `instanceof peer` array after the transformation.

```
(x instanceof rep <type> T[])
```

Listing 3.22: `instanceof rep` array before the transformation.

```
( (x instanceof T[]) &&
    (x.$owner == this) &&
    (x.$type == <type>) )
```

Listing 3.23: `instanceof rep` array after the transformation.

```
(x instanceof readonly <type> T[])
```

Listing 3.24: `instanceof readonly` array before the transformation.

```
( (x instanceof T[]) &&
    (x.$type == <type>) )
```

Listing 3.25: `instanceof readonly` array after the transformation.

If the Universe type of the array is the same for the static type of the object and the right hand side of the `instanceof` expression, the owner check can be omitted. The same holds if the static Universe type of the array variable is not `readonly`.

3.2.3 Casts

If the element type does not change when casting an array, it is equivalent to casting an object and can be handled just like the transformations described in chapter 3.1.3. The same holds if it changes from `peer` to `readonly`, because then it is an upcast and upcasts are always allowed. Finally if it does change from `readonly` to `peer`, the following transformation applies.

```
readonly readonly S[] x;
peer peer T[] y = (peer peer T[]) x;
```

Listing 3.26: An array cast to `peer peer` before the transformation.

```
S [] x;
if ( (x.$owner != this.$owner) ||
        (x.$type != peer) )
    throw new ClassCastException();
T[] y = (T[]) x;
```

Listing 3.27: An array cast to `peer peer` after the transformation.

```
readonly readonly S[] x;
rep peer T[] y = (rep peer T[]) x;
```

Listing 3.28: An array cast to `rep peer` before the transformation.

```
S [] x;
if ( (x.$owner != this) ||
        (x.$type != peer) )
    throw new ClassCastException();
T [] y = (T []) x;
```

Listing 3.29: An array cast to `rep peer` after the transformation.

If the Universe type of the array is the same for the static type of the object and the right hand side of the `instanceof` expression, the owner check can be omitted. The same holds if the static Universe type of the array variable is not `readonly`.

3.2.4 Assignments to array elements

For any assignment to an array component, we must assure that we are not writing a `readonly` or `rep` object into an array of `peer` elements.

```
<type> readonly S x[];
x[0] = y;
```

Listing 3.30: Assignment to an array component before the transformation.

```
S x [];
if ( (x.$type != readonly) &&
        (x.$owner != y.$owner) )
    throw new ArrayStoreException();
x[0] = y;
```

Listing 3.31: Assignment to an array component after the transformation.

Where `y` may be of type `rep`, `peer` or `readonly` `T` or `S` and `<type>` may be `peer` or `rep`. If it were `readonly`, the compiler would complain about the write access to a `readonly` variable.

Chapter 4

Implementation

The original idea was to use the standard `java.util.Hashtable` and generate a unique hashcode for every object using `System.identityHashCode()`. An entry in the hashtable would then use this hashcode as the key and the hashcode of the owner as the data item.

4.1 The owner Hashtable

Unfortunately `System.identityHashCode()` is not really unique as it is not coupled to the object identity. It is not associated to something like the memory location of the object. Apart from that it might be that the owner of an object has already been collected by the garbage collector, while the object itself is still referenced and therefore still alive. This way even a hashcode coupled to memory locations could produce duplicates for our table[1]. We then need the entry for the already deleted owner to remain in the hashtable. Now if a newly created object is put to the now free memory location of the owner, it would have the same hashcode and would be considered the owner of the old object, whose owner has been deleted. Using a simple counter instead of a hashcode is no solution since we need to be able to reproduce the hashcode.

Another problem was the performance issue. Using a `java.util.Hashtable` with `int` keys (and `int` values) involves boxing the `int` into an `Integer` object for every operation, thus costing time and space. Additional space is also needed for internal objects of the hashtable.

Now here are the requirements we think a hashtable has to meet, to be usable for this project:

- It has to allow multiple entries with the same key. Possibly by treating it the same way as a normal collision. And on lookups it has to be able to find the entry for the exact object we are looking for and must not rely on the hashcode, because we cannot guarantee the uniqueness of the hashcodes.
- It has to consume as little space as possible and create as few internal objects as possible. Not only to save space but also to minimize the overhead of CPU time needed for the runtime checks.
- It has to be able to detect when a user object is collected and must then free the hashtable entry if it is no longer used, which is the case if the universe it owns is empty.

We decided to use a stripped version of the `IntHashtable` from ACME[2] to get rid of the overhead caused by boxing and unboxing `int` values. Then I added the ability to store multiple entries with the same key in the hashtable. Finally the identity is checked by using a `WeakReference` to the

[1] Also on 64 Bit machines it is impossible to provide a unique mapping of memory locations to 32 Bit `int` values.

[2] http://www.acme.com/java/software/Acme.IntHashtable.html

original object and then using object identity to make sure the entry is the right one. Whenever an object is looked up in the hashtable it is still alive, so this will work. Now to compare the owner of two objects, one can simply check if the entries of the owners are identical which can be done with object identity. This also works when the owner object itself has already been garbage collected.

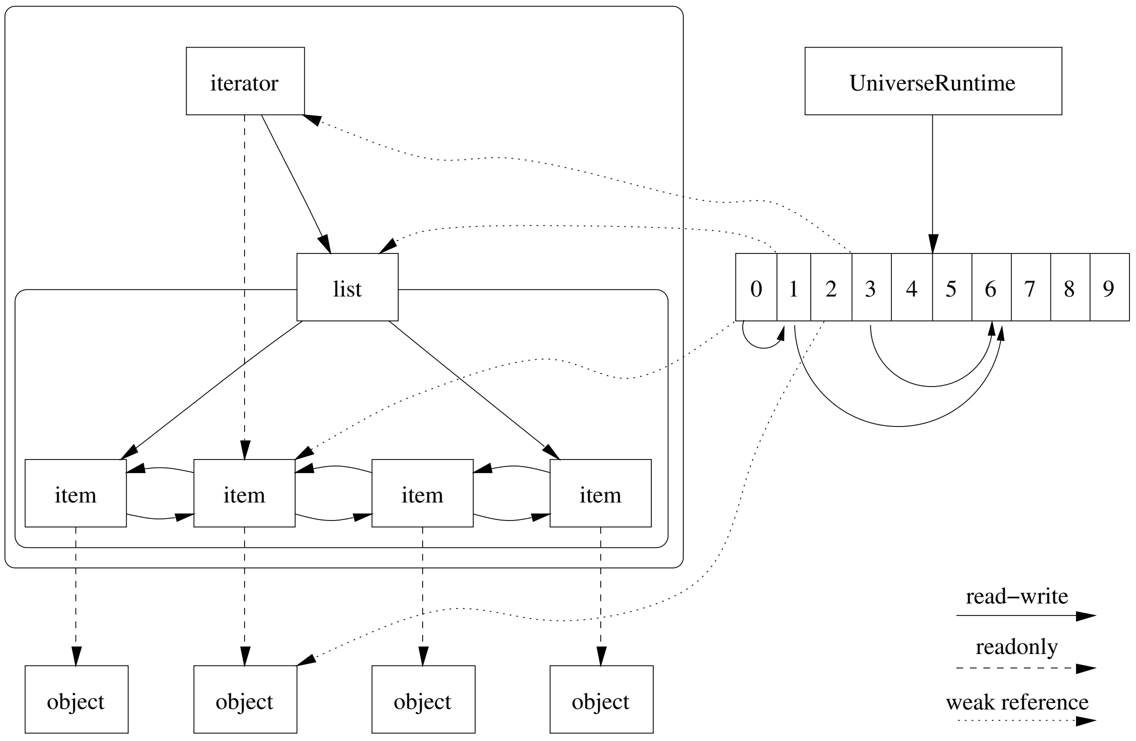


Figure 4.1: The structure of the hashtable.

Figure 4.1 shows the hashtable and the weak references to some of the objects[3]. The references between the hashtable entries link from an objects entry to the entry of the owner of that object.

Now for example if we want to check whether the iterator and the list head belong to the same universe, we lookup both of them in the hashtable and compare the hashtable entries of the respective owners using object identity. This is equivalent to comparing the owners themselves, but saves us one level of indirection and it also works when the owner has already been collected.

To track if an objects universe is empty or not, a counter for the number of objects in the universe is maintained.

4.1.1 The runtime classes

Here is a short overview over the runtime classes, located in the `org.multijava.universes.rt` package.

- `UniverseRuntime.java`: The main class that provides access to implementation and policy through fields and initializes them when it is loaded.
- `UrtImplementation.java`, `UrtPolicy.java`: Interfaces for the implementations and policies.

[3]To avoid confusion because of the sheer mass of references, not all weak references are shown.

- `UrtDebug.java`, `UrtDefaultImplementation.java`, `UrtDummy.java`, `UrtVisualizer.java`: Implementations, see chapter 4.3.2.1 for details. They are located in the `impl` subpackage.
- `UrtHashtable.java`: The Hashtable explained above. It goes along with the default implementation `UrtDefaultImplementation.java` but by having it in a separate class it can also be used by other implementations.
- `UrtDefaultPolicy.java`, `UrtDummy.java`, `UrtRelaxed.java`, `UrtStrict.java`: Policies, see chapter 4.3.2.2 for details about these and chapter 4.1.1.1 for a description on what can be defined through policies. They are located in the `policy` subpackage.

See the Javadoc comments in the listings C.4 and C.5 to get more details on how the implementation works.

4.1.1.1 The policy classes

The idea of splitting the runtime classes into implementation and policy classes makes it easier to define the behaviour of the runtime checks without having to change the implementation. Here is a short description of the functions a policy provides and what they are used for.

- `boolean isExternalPeer()` defines if objects created in legacy code should be handled as `peer`.
  The default policy returns `true`.
- `Object getNativeOwner(Object obj)` defines the owner that is used for Universe aware objects that are created outside of Universe aware code. See chapter 4.2.2 for details on this. There are many possible approaches. One could for example use a virtual set for legacy objects or the root set.
  The default policy returns `null` and relies on the implementation to do something reasonable, as the implementation knows more about the structure of the object store. The default implementation just registers the new object as `peer` to the last Universe aware context object in the call hierarchy, if the policy returns `null`.
- `Object getCollectedOwner(Object obj)` defines what to return when a reference to an objects owner is requested but the owner has already been collected.[4]
  The default policy returns `null`.
- `void illegalCast()` handles illegal casts of Universe objects.
  The default policy throws a `ClassCastException`.
- `void illegalArrayStore()` handles illegal stores to arrays of object type.
  The default policy throws an `ArrayStoreException`.

4.2 Various aspects of this solution

4.2.1 Initialization

The hashtable of course has to be initialized before objects can be put in there. This is done in the static initialization block of the `UniverseRuntime` class. So the hashtable will be initialized, when the first runtime check related function is called.

[4] A reference to an objects owner can be requested using the `getOwner()` function that every implementation class needs to implement.

4.2.2 Field initialization and object creation in constructors

Another problem is the creation of new objects and the use of casts and the `instanceof` operator in the constructor[5]. If the object is registered to the hashtable after it has been created, like it is pointed out by the transformations in chapter 3.1.1, registering objects created in the constructor or for field initialization will fail. Their registration to the hashtable would have to be relative to the current object, which is not registered yet at that point.

Registering the object between the execution of the `new` operator and the constructor call is not allowed as stated in section 4.9.4 of the JVM Specification [Sunb]. Therefore the transformation has been slightly changed to fit this situation:

```
class T {
    public T () {
        // ...
    }
}

// ...
new peer T();
```

Listing 4.1: `new peer` before the transformation

```
class T {
    public T () {
        if ( this.getClass() == Thread.
            currentThread().$nextClass )
            this.$owner = Thread.currentThread
                ().$nextOwner;
        else
            this.$owner = $defaultOwner;
        // ...
    }
}

// ...
Thread.currentThread().$nextOwner = this.
    $owner;
Thread.currentThread().$nextClass = T.class;
// create object and call constructor
T tmp = new T();
tmp.$owner = this.$owner;
```

Listing 4.2: `new peer` after the transformation

For simplicity and to make it easier to compare this transformation with the original transformation in chapter 3.1.1, the same simplified notation has been used, instead of the calls to the owner hashtable functions.[6]

Before the creation of the object, a reference to the current contexts owner, as well as the type and the Universe modifier of the object to be created, are saved in the hashtable. This data is stored with the hashtable entry for the current thread, to make it easy to find it again. In the transformation above the virtual fields `$nextOwner` and `$nextClass` are used for this purpose.

In the constructor then, before anything else happens, the new object is registered to the hashtable, by calling a hashtable function. The type of the new object is needed to avoid using the wrong data for the registration to the hashtable. This could happen if a new legacy object is created, which will, of course, be registered only after the new statement. Now if this legacy object creates a Universe aware object in its constructor, the data stored for the legacy Object would be used. Comparing the intended and the actual classtype can reveal that and help using the correct registration method, which can be defined by the policy.

Unfortunately the additional attempt to register the new object, after the constructor has been called, cannot be removed. If the object created is not Universe aware, it has to be registered after the constructor call and since no additional data for Universe aware classes are stored in the classfile yet, it cannot be omitted for these either. At least not at the moment.

[5] also indirectly by calling functions in the constructor
[6] The bytecode analysis in chapter 5.2 shows the real calls to the hashtable, as generated by the compiler.

4.2.3 Static functions

Static functions are special, since no `this` reference exists, which is the owner of the current universe in normal functions. The implemented solution runs static functions in the same context as the calling method. The `main()` function is of course special and runs in the root context.

To make this possible, before any call to a static function, the current context is pushed on top of a stack that is stored with the current threads hashtable entry. Within static methods the context is then fetched from the hashtable, while normal functions still use the `this` reference. And when the static function returns, the topmost element of the stack is removed again. This way a hierarchy with multiple levels is possible.

```
class foo {
    public static void bar () {
        peer Object o = new Object();
    }

    public void caller () {
        foo.bar();
    }
}
```

Listing 4.3: `peer` call to a static method before the transformation

```
class foo {
    public static void bar () {
        readonly Object context = Thread.
            currentThread().$contextStack.peek()
            ;
        Object o = new Object();
        o.owner = $context;
    }

    public void caller () {
        Thread.currentThread().$contextStack.
            push(this);
        bar();
        Thread.currentThread().$contextStack.
            pop();
    }
}
```

Listing 4.4: `peer` call to a static method after the transformation

```
class foo {
    public static void bar () {
        peer Object o = new Object();
    }

    public void caller () {
        rep foo.bar();
    }
}
```

Listing 4.5: `rep` call to a static method before the transformation

```
class foo {
    public static void bar () {
        readonly Object context = Thread.
            currentThread().$contextStack.peek()
            ;
        Object o = new Object();
        o.$owner = context.$owner;
    }

    public void caller () {
        tmp = new Object();
        tmp.$owner = this;
        Thread.currentThread().$contextStack.
            push(tmp);
        foo.bar();
        Thread.currentThread().$contextStack.
            pop();
    }
}
```

Listing 4.6: `rep` call to a static method after the transformation

Static functions can be called either on a reference or on a typename. For `peer` references and normal calls on the typename, the current object is used as context object inside the static function. For `rep` references, the referenced object is used, if the reference is not the `null` reference. Otherwise and for `rep` calls on a typename, a temporary `rep` object is created and used.

```
class foo {
    public static void bar () {
        // ...
    }

    public void caller () {
        rep foo.bar();
    }
}
```

Listing 4.7: Example on how to call a static function `rep` on a typename

For efficency, the current context is only pushed onto the stack if the call to the static function is within a normal function. Within static functions it would just duplicate the topmost stack element and can thus be omited.

Static initialization blocks and static fields, initialized using the `new` operator, are executed when the class is loaded, which is determined by the virtual machine. Therefore we decided to run them in the root context, just like the `main()` function, to get a predictable behaviour.

4.2.4 Pure methods

Pure methods do not have to be handled special. They have a context universe so all tests should work.

4.2.5 Legacy code

Having an initialization that does not rely on the user, allows to use libraries written in Universe Java from within legacy code, without having to take care of the initialization.

Passing arrays to legacy code is still an open issue, since access to arrays cannot be restricted using getters and setters, like it is possible for objects. Therefore legacy code can break invariants by storing objects of an invalid Universe type into arrays.

The creation of Universe aware objects from outside of Universe aware code should be handled fine but things get complicated when legacy code doing such things is used.

The same holds for static Universe Java functions called from legacy code. It works, but practice will have to show if the implemented solution is good or not.

4.3 Usage

4.3.1 Compiling and running Universe programs with runtime check support

Simply compile your program using the `--universes` switch to mjc:

```
# java org.multijava.mjc.Main --universes hello.java
```

You can then run it like any normal Java program, if the `org.multijava.universes.rt` package containing the runtime classes is in the classpath:

```
# java hello
```

4.3.2 Using alternative implementations of the runtime classes

The runtime classes are divided in implementation and policy classes. At runtime, one of each is needed. The implementation class is responsible for registering all objects to the hashtable and do the checks, while the policy class defines the behaviour of the runtime checks. For example whether invalid casts should generate an exception or just a warning.[7]

Both, the implementation and the policy class, can be replaced by alternative versions independently.

4.3.2.1 Implementation classes

Instead of using the default implementation `UrtDefaultImplementation`, you may use an alternative version. Either one that also comes with the MultiJava Compiler or a self written one.

MultiJava is delivered with four implementation classes:

- `UrtDefaultImplementation` (listing C.4) is the default one, provides all tests and uses a hashtable as described above.
- `UrtDebug` (listing C.6) prints out some debugging information, which may be helpful, if you want to see what tests are done. Apart from that it behaves just like `UrtDefaultImplementation`.
- `UrtDummy` (listing C.7) does not do anything. It returns `true` for all tests and `null` for all functions returning objects. It is usefull to "disable" the runtime checks for a program without recompiling it.
- `UrtVisualizer` (listing C.8) writes a dot-file[8] that can be used to visualize the object store. To use this implementation, you need to specify the filename for the output as an additional argument to the VM on the command line: `-DUrtOutfile=<filename>`

To use an alternative implementation, simply specify it as argument to the VM on the command line. To use `UrtDebug` for example you can call your program like this:

```
# java -DUrtImplementation=org.multijava.universes.rt.impl.UrtDebug hello
```

If you specify an invalid implementation class, the program will print a warning and fall back to the default, which is `UrtDefaultImplementation`.

4.3.2.2 Policy classes

Instead of using the default policy `UrtDefaultPolicy`, you may use an alternative version. Either one that also comes with the MultiJava Compiler or a self written one.

MultiJava is delivered with four policy classes:

- `UrtDefaultPolicy` (listing C.9) is the default one and is intended to integrate as smooth as possible. It throws errors on illegal casts and illegal write accesses to array elements and handles external non-Universe objects as `peer` objects.
- `UrtDummy` (listing C.10) does not do anything. It does not throw errors and handles external non-Universe objects as `peer` objects.
- `UrtRelaxed` (listing C.11) does not throw an exception on illegal casts and illegal write accesses to array elements but just displays an error message.
- `UrtStrict` (listing C.12) handles external objects as `readonly`, which is better with regards to the Universe type system, but bad for compatibility with legacy code like the Java SDK, since it pretty much disallows the use of any function in the SDK that takes parameters of reference type. This is because any variable, parameter or object that is not annotated is considered to have Universe type `peer`.

[7]See chapter 4.1.1.1 for a more insight listing of the behaviours that can be tweaked using policies.

[8]dot ([GKNV93]) is part of the Graphviz package (http://www.graphviz.org/) and makes "hierarchical" or layered drawings of directed graphs.

To use an alternative policy, simply specify it as argument to the VM on the command line. To use `UrtStrict` for example you can call your program like this:

```
# java -DUrtPolicy=org.multijava.universes.rt.policy.UrtStrict hello
```

If you specify an invalid runtime class, the program will print a warning and fall back to the default, which is `UrtDefaultPolicy`.

4.3.3 Writing an own runtime class

All you need to do is make sure it implements `UrtImplementation` (listing C.2) if it is an implementation class or `UrtPolicy` (listing C.3) if it is a policy class.

4.3.4 Distributing programs that use Universe runtime checks

If you distribute your program, you need to deliver it along with the runtime classes in the `org.multijava.universes.rt` package. The easiest way to assure this is by using the JAR-file that can be generated with the `universe-rt-jar` target in the toplevel Makefile of the MultiJava compiler.

4.4 Testcases

I have created a total of 56 testcases to check whether `instanceof`, casts for objects and arrays, write access to an array element, exceptions, static functions, interaction with multiple (legacy Java) threads and legacy code in general work fine. The testcases fit into the existing relaxation tests of the MultiJava compiler and can be executed and compared to the expected results automatically using a target in the Makefile.

Listings 4.8 - 4.11 show the testcases for `instanceof peer`. The testcases for the other checks are very similar. For every check there is at least one test that should succeed and one that should fail. Except for casts to `readonly` and `instanceof readonly` which should always succeed.

```
package org.multijava.mjc.testcase.universes.runtime;

public class instanceof_peer_success1 {
    public static void main (String[] args) {
        new foo().bar();
    }
}

public class foo {
    public void bar () {
        readonly Object o = new peer Object();
        System.out.println(o instanceof peer Object);
    }
}
```

Listing 4.8: Test if `instanceof peer` evaluates to `true` for `peer` objects.

```
package org.multijava.mjc.testcase.universes.runtime;

public class instanceof_peer_success2 {
    public static void main (String[] args) {
        new foo().bar();
    }
}

public class foo {
    public void bar () {
        readonly Object o = external.getObjectStatic();
        System.out.println(o instanceof peer Object);
    }
}
```

Listing 4.9: Test if `instanceof peer` evaluates to `true` for external (legacy Java) objects.

```
package org.multijava.mjc.testcase.universes.runtime;

public class instanceof_peer_fail1 {
    public static void main (String[] args) {
        new foo().bar();
    }
}

public class foo {
    public void bar () {
        readonly Object o = new rep Object();
        System.out.println(o instanceof peer Object);
    }
}
```

Listing 4.10: Test if `instanceof peer` evaluates to `false` for `rep` objects.

```
package org.multijava.mjc.testcase.universes.runtime;

public class instanceof_peer_fail2 {
    public static void main (String[] args) {
        new foo().bar();
    }
}

public class foo {
    public void bar () {
        functions f = new functions();
        readonly Object o = f.getReadonlyObject();
        System.out.println(o instanceof peer Object);
    }
}
```

Listing 4.11: Test if `instanceof peer` evaluates to `false` for `readonly` objects.

4.5 The changes to the MultiJava Compiler

Here is a short overview of the files of the MultiJava [MJC] Compiler that have been modified and a quick summary what has been done for each file. All modifications have been marked with my id "scdaniel".

- `CArrayType.java`: Changed the policy for Universe Java to allow downcasts if the runtime checks are turned on.
- `Constants.java`: Added two new constants: the names for the temporary variables.
- `CUniverseRuntimeHelper.java`: New class to minimize code duplication.
- `JArrayAccessExpression.java`: Only display compiler warnings for covariant array access if runtime checks are disabled.
- `JAssignmentExpression.java`: Generate the bytecode for righthandsides that are of type `JNewObjectExpression` or `JNewArrayExpression`.

- `JBlock.java`: Generate bytecode for static initializers.
- `JCastExpression.java`: Generate the bytecode for the runtime test.
- `JConstructorBlock.java`: Generate bytecode for the call to `setOwner` that is needed in constructors (see chapter 4.2.2).
- `JExpression.java`: Base implementation of the general methods called by the parent nodes `JNewObjectExpression` and `JNewArrayExpression`. This usage of inheritance simplifies the code a lot.
- `JFieldDeclaration.java`: Generate the bytecode for field initializers that are of type `JNewObjectExpression` or `JNewArrayExpression`.
- `JInstanceofExpression.java`: Generate the bytecode for the runtime test.
- `JMethodCallExpression.java`: Generate the bytecode to push the current context object onto the context stack before calling a static function, as well as removing it again when the static method returned.
- `JMethodDeclaration.java`: Generate the bytecode for static initializers.
- `JNewArrayExpression.java`: Generate bytecode to register new arrays in a general method called from the parent node. Also generate bytecode to register anonymous arrays to the hashtable.
- `JNewObjectExpression.java`: Generate bytecode to register new objects in a general method called from the parent node. Also generate bytecode to register anonymous objects to the hashtable.
- `JVariableDeclarationStatement.java`: Generate the bytecode for variable initializers that are of type `JNewObjectExpression` or `JNewArrayExpression`.
- `Main.java`: Provide functions to temporarily disable Universe type checks. I needed that to be able for example to register new `rep` objects to the hashtable. With universe type checks enabled this will give an error, because the hashtable functions are written in legacy Java and therefore take `peer` arguments. The hashtable is not written in Universe Java to avoid confusion and because it uses external JDK functions so an implementation in Universe Java could not use `readonly` parameters either.
- `UniverseRuntime.java`, `UrtImplementation.java`, `UrtPolicy.java`, `UrtDebug.java`, `UrtDefaultImplementation.java`, `UrtDummy.java`, `UrtHashtable.java`, `UrtVisualizer.java`, `UrtDefaultPolicy.java`, `UrtDummy.java`, `UrtRelaxed.java`, `UrtStrict.java`: New classes used at runtime for the tests.
  These have been put in a separate package `org.multijava.universes.rt`.

The changes in `JAssignmentExpression.java`, `JFieldDeclaration.java` and `JVariableDeclarationStatement.java` could have been omitted for the cost of using a temporary variable for every new object or array. The implementation in the compiler is a little bit more complex now, but the costs at runtime should be lower thanks to that.

Chapter 5

Evaluation

5.1 Benchmarks

How good is this solution? To give a rough idea of its performance, I "benchmarked" it with three tests.

All tests have been done under GNU/Linux on a 3GHz Intel Pentium 4 machine with 1GB memory. The running times are mean values of three runs measured with the standard Unix "`time`" command. The memory consumption has been measured with JMP [Olo], a memory profiler for java. I have decided to show only the relevant parts of the memory consumption, using human readable units, instead of including the whole statistic. All tests are designed in a way that makes the relevant parts consume enough memory to be part of this partial statistics[1].

The Makefile used to run the tests is found in listing C.13 in the Appendix.

5.1.1 Objecttest

The first test is basically a loop and just creates 1'000'000 instances of `java.lang.Object` (listing C.14).

Universe runtime checks enabled	no	yes
runtime	0.43s	7.38s
total amount of memory consumed	11.65 MB	79.06 MB
`org.multijava.universes.rt.impl.UrtWeakReference`	-	30.52 MB
`org.multijava.universes.rt.impl.UrtHashtableEntry`	-	30.52 MB
`java.lang.Object[]`	3.84 MB	10.21 MB
`java.lang.Object`	7.63 MB	7.63 MB

The results for this test look very bad. Almost twenty times slower and eight times as much memory consumption. But the downside of this test is its simplicity. Instances of `java.lang.Object` seem to consume little to no space and only creating objects without doing something else is not a very realistic scenario.

5.1.2 Stringtest

The next test creates 100'000 random strings of length 256 (listing C.15).

[1]in other words: there are enough loop iterations

Universe runtime checks enabled	no	yes
runtime	4.15s	4.83s
total amount of memory consumed	53.23 MB	60.53 MB
`char[]`	50.41 MB	50.41 MB
`org.multijava.universes.rt.impl.UrtWeakReference`	-	3.05 MB
`org.multijava.universes.rt.impl.UrtHashtableEntry`	-	3.05 MB
`java.lang.String`	2.30 MB	2.30 MB
`java.lang.Object[]`	0.40 MB	1.60 MB

Now this looks a lot better. Approximately 16 percent overhead on the runtime and 14 percent overhead on memory consumption. The big memory part here belongs to the strings respectively the char arrays.

5.1.3 Listtest

This test uses a doubly linked list and randomly inserts, deletes and modifies list items. Each operation is executed about 330'000 times (listings C.16, C.17 and C.18).

Universe runtime checks enabled	no	yes
runtime	3m20.08s	3m48.31s
total amount of memory consumed	1.32 MB	5.11 MB
`org.multijava.universes.rt.impl.UrtWeakReference`	-	2.06 MB
`org.multijava.universes.rt.impl.UrtHashtableEntry`	-	1.89 MB
`java.lang.Object[]`	0.02 MB	0.42 MB
`item`	0.52 MB	0.28 MB
`java.lang.Integer`	0.60 MB	0.27 MB

This test shows probably best how big the expected overhead for a program using the Universe runtime checks is. There is a little overhead of about 14 percent on the runtime and almost three times the memory consumption. But again, the data stored in the list is a simple `int`, boxed into an `Integer` object. Choosing something more realistic would certainly make this better. I decided to use a simple object again and not something like the random strings used in the previous test, to find the runtime overhead that is caused by the list operations. It is obvious that repeating the step towards bigger data sizes, from the first test to the second one, for this test would result in less memory overhead again.

5.1.4 Summary

The overhead on the runtime is relatively small and should be acceptable. If a program has to be really fast, Java is probably the wrong choice anyway.

Memory consumption is a bit more critical, but should be acceptable too for real programs that actually handle some data and not only `Integer` objects.

Please be aware of the fact that benchmarking is a very complex field and the above tests give only a rough overview over the performance of the hashtable. Serious benchmarking would certainly require much more than just three simple test programs running on a single machine.

5.2 Code comparison

Listings C.19 and C.20 show the effect of the transformations to an example program. This also shows how the runtime classes are actually used and how the code is transformed. The decompiled sourcecode in listing C.20 has been generated using JAD [Kou], a Java decompiler.

Special things to see here are the call to `setConstructorData()` before every object creation, which is needed for field initialization and code in constructors of Universe aware classes, as described in chapter 4.2.2. The counterpart of this function call is then found in the constructor of the decompiled `test` class in listing C.20: a call to `setOwner()` right at the beginning.

After each object creation an additional call to either `setOwnerPeer()` or `setOwnerRep()` is made. This is needed to create objects of legacy classes and cannot be omitted for objects of Universe aware classes, because the compiler does not know which classes are Universe aware and which are not.

Another difference to the abstract transformations of chapter 3 is that the actual transformations for `instanceof` and casts use temporary variables to avoid evaluating expressions twice. For the array store check the use of temporary variables is avoided by first storing the object into the array and afterwards checking if it was legal.

Before a static function is called, the current context object is pushed onto the context stack using `setContext()` and after the function call, the stack is reset again by calling `resetContext()`. Inside static functions `getContext()` is used to retrieve the context object, as no `this` reference exists.

For static initializers, the root context is used.

5.3 Conclusion and open issues

There are still a few open issues, mainly in the field of interaction with legacy code:

- Universe Java objects created from within legacy code are by default registered relative to the last object created in Universe Java code. Practice will have to show if this solution is good or not.
- Universe aware static functions called from legacy code use the context on top of the stack. It would probably be better to recognize this case and handle it separately, as it is done for Universe objects created from legacy code, but it is a little harder for this case.

Apart from that I think the solution works well and is an acceptable implementation of the runtime checks for the Universe type system.

Here is a small list of open issues that are not part of this project but related to the work I have done:

- Reflection is a topic related to runtime checks. Currently no Universe specific reflection support exists.
- Anonymous classes are not yet covered by the Universe Compiler and are therefore not registered to the hashtable.
- Passing arrays to legacy code is dangerous since legacy code can store objects of an invalid Universe type into arrays and therefore break invariants.

Bibliography

[BCE] BCEL - byte code engineering library.
http://bcel.sourceforge.net/.

[DM04] W. Dietl and P. Müller. Exceptions in ownership type systems. In E. Poll, editor, *Formal Techniques for Java-like Programs*, pages 49–54, 2004.

[DM05] W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology (JOT)*, 2005. To appear.

[DMS05] W. Dietl, P. Müller, and D. Schregenberger. *Universe Type System Quick-Reference*, May 2005. Part of the MultiJava distribution.

[Frea] Free Software Foundation, Inc. GNU general public licence.
http://www.gnu.org/copyleft/gpl.html.

[Freb] Free Software Foundation, Inc. GNU lesser general public licence.
http://www.gnu.org/copyleft/lesser.html.

[GKNV93] Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Kiem-Phong Vo. A technique for drawing directed graphs. 19(3):214–230, May 1993.
http://www.research.att.com/sw/tools/graphviz/TSE93.pdf.

[GNU] The GNU classpath project.
http://www.gnu.org/software/classpath/classpath.html.

[Kou] Pavel Kouznetsov. JAD - the fast Java decompiler.
http://www.kpdus.com/jad.html.

[Mey] John Meyer. Jasmin.
http://mrl.nyu.edu/~meyer/jvm/jasmin.html.

[MJC] The multijava project.
http://multijava.sourceforge.net/.

[MPH01] P. Müller and A. Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical Report 279, Fernuniversität Hagen, 2001.

[Olo] Robert Olofsson. JMP - Java memory profiler.
http://www.khelekore.org/jmp/.

[Rea] Patrik Reali. GNU classpath VM integration guide.
http://www.gnu.org/software/classpath/docs/vmintegration.html.

[Suna] Sun Microsystems, Inc. The Java language specification.
http://java.sun.com/docs/books/jls/second_edition/html/jTOC.doc.html.

[Sunb] Sun Microsystems, Inc. The Java virtual machine specification.
http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html.

Appendix A

Sun Licence

Section D of the Supplemental Licence Terms of the “Sun Microsystems, Inc. Binary Code License Agreement for the JAVATM 2 SOFTWARE DEVELOPMENT KIT (J2SDK)”[1]:

> D.Java Technology Restrictions. You may not modify the Java Platform Interface (”JPI”, identified as classes contained within the ”java” package or any subpackages of the ”java” package), by creating additional classes within the JPI or otherwise causing the addition to or modification of the classes in the JPI. In the event that you create an additional class and associated API(s) which (i) extends the functionality of the Java platform, and (ii) is exposed to third party software developers for the purpose of developing additional software which invokes such additional API, you must promptly publish broadly an accurate specification for such API for free use by all developers. You may not create, or authorize your licensees to create, additional classes, interfaces, or subpackages that are in any way identified as ”java”, ”javax”, ”sun” or similar convention as specified by Sun in any naming convention designation.

[1]The complete licence can be found on `http://java.sun.com/j2se/1.4.2/j2sdk-1_4_2_05-license.txt`

Appendix B

Changes since the initial submission

- Static functions should now be handled correctly, as described in chapter 4.2.3.
- The runtime classes have been moved into a separate package and a JAR-file is provided to be delivered along with Universe Java programs. Furthermore the licencing of the runtime classes has been changed from GPL [Frea] to LGPL [Freb] to avoid forcing users of the Universe type system to release their software under the GPL as well.
- The interaction with external code has been improved as described in chapter 4.2.5.
- The runtime classes should now be threadsafe. Also Threads created in legacy code should be handled correctly and not crash the implementation when trying to save or load data in a Threads hashtable entry.
- The bug that generated compiler errors, complaining about missing methods, has been fixed.
- A couple of new testcases have been created to test static functions, interaction with multiple (legacy Java) threads and legacy code in general.

Appendix C

Sourcecode

C.1 Runtime Classes

C.1.1 Initialization

```
/*
 * This file is part the Universe Runtime Classes.
 *
 * Copyright (C) 2003−2005 Swiss Federal Institute of Technology Zurich
 *
 * Part of mjc, the MultiJava Compiler.
 *
 * Copyright (C) 2000−2005 Iowa State University
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111−1307 USA
 *
 * $Id$
 */

package org.multijava.universes.rt;

import org.multijava.universes.rt.impl.*;
import org.multijava.universes.rt.policy.*;

/**
 * This Class manages the dynamic type checking for the universe type
 * system.
 *
 * This class just defines to static fields and a static initializer where
 * the real runtime classes are instantiated.
 *
 * There are two types of runtime classes: the implementation class that takes
 * care of the objects (the default implementation (UrtDefaultImplementation)
 * uses an optimized hashtable) and the policy class (default: UrtDefaultPolicy)
 * that defines what happens when dynamic checks fail etc.
 *
 * To create your own implementations of one of these you just need your class
 * to implement the respective Interface:
 *  - UrtImplementation
 *  - UrtPolicy
 *
 * @author scdaniel
 */
public class UniverseRuntime {
    /**
     * this field stores at runtime a reference to an instance of a
     * class to be used as handler (hashtable) for the RuntimeChecks.
     */
    public static UrtImplementation handler;

    /**
     * this field stores at runtime a reference to an instance of a
     * class to be used as policy for the RuntimeChecks.
     */
    public static UrtPolicy policy;

    /**
     * Static initializer.
     * Instantiates an object of the class specified by the JVM
     * command-line switch -DUrtImplementation=<class> and stores it into
     * the static field handler and instiantiates an object of the class
     * specified by -DurtPolicy=<class> and stores it into the static field
     * policy.
     * Defaults to impl.UrtDefaultImplementation and policy.UrtDefaultPolicy
     * if nothing is specified or instantiation fails.
     */
    static {
        /* get name of implementation class */
        String runtimeClass = System.getProperty("UrtImplementation");
        if ( runtimeClass != null ) {
            /* try to create an instance of this class */
            try {
                handler = (UrtImplementation)
                            Class.forName(runtimeClass).newInstance();
            } catch ( Exception e ) {
                System.err.println("WARNING: the specified Universe Runtime Implementation
                    Class (" +
                        runtimeClass +
                        ") could not be instantiated, using the default instead.");
                System.err.println(e);
                handler = new UrtDefaultImplementation();
            }
        } else
            handler = new UrtDefaultImplementation();

        /* get name of policy class */
        runtimeClass = System.getProperty("UrtPolicy");
        if ( runtimeClass != null ) {
            /* try to create an instance of this class */
            try {
                policy = (UrtPolicy)
                            Class.forName(runtimeClass).newInstance();
            } catch ( Exception e ) {
                System.err.println("WARNING: the specified Universe Runtime Policy Class (" +
                        runtimeClass +
                        ") could not be instantiated, using the default instead.");
                System.err.println(e);
                policy = new UrtDefaultPolicy();
            }
        } else
            policy = new UrtDefaultPolicy();
    }
}
```

Listing C.1: The UniverseRuntime class that loads the actual implementation and policy classes.

C.1.2 Interfaces

```
/*
 * This file is part the Universe Runtime Classes.
 *
 * Copyright (C) 2003-2005 Swiss Federal Institute of Technology Zurich
 *
 * Part of mjc, the MultiJava Compiler.
 *
 * Copyright (C) 2000-2005 Iowa State University
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
 *
 * $Id$
 */

package org.multijava.universes.rt;

/**
 * This interface has to be implemented by any class that is meant to be
 * used as implementation for the Universe Runtime Checks.
 *
 * The default implementation (UrtDefaultImplementation) uses an optimized
 * hashtable to manage the objects and their relations.
 *
 * @author scdaniel
 */
public interface UrtImplementation {
    /**
     * Registers the ownership relation of obj and owner in the hashtable.
     * (owner is the owner of obj)
     *
     * @param obj the object
     * @param owner it's owner
     */
    abstract void setOwnerRep (Object obj, Object owner);

    /**
     * Registers the ownership relation of obj and current in the hashtable.
     * (they both share the same owner)
     *
     * @param obj       the object
     * @param current   the owner of the current universe when obj is created
     */
    void setOwnerPeer (Object obj, Object current);

    /**
     * Saves the current object and the modifier for the object being created in
     * its context in the current threads hashtable entry.
     * This information is later retrieved by setOwner (which is called in the
     * constructor) to set the owner of newly created objects before they
     * create any other objects in the constructor.
     *
     * @param currentObject the object that owns the current universe
     * @param objectClass   the class of the object that is to be created
     * @param modifier      the modifier that applies to the object to be created
     *                      (one of MjcTokenTypes.LITERAL_peer, MjcTokenTypes.LITERAL_rep)
     */
    void setConstructorData (Object currentObject, Object objectClass, int modifier);

    /**
     * Sets the owner of the current object by using the data stored in the
     * the current threads hashtable entry.
     * This function should be called in the constructor, before any object
     * (including field initializers ) is created.
     *
     * @param o the current object
     */
    void setOwner (Object o);

    /**
     * Registers the ownership relation of the array obj and the object owner
     * in the hashtable. (owner is the owner of obj)
     * And stores the universe type of the array elements in the hashtable entry
     * that is used for the object.
     *
     * @param obj   the object
     * @param owner it's owner
     */
    void setArrayOwnerRep (Object[] obj, Object owner,
            int arrayElementType);

    /**
     * Registers the ownership relation of the array obj and the object current
     * in the hashtable. (they both share the same owner)
     * And stores the universe type of the array elements in the hashtable entry
     * that is used for the object.
     *
     * @param obj       the array
     * @param current   the owner of the current universe when obj is created
     */
    void setArrayOwnerPeer (Object[] obj, Object current,
            int arrayElementType);

    /**
     * Checks if the array specified has *exactly* the elementType specified.
     *
     * For ArrayStores, we need to know whether it is really readonly. That's
     * why it does not work the same as instanceof and therefore returns only
     * true if the actual element type is equal to elementType.
     *
     * @param o             the array to check
     * @param elementType the element type to check for
     * @return true if the elementType of o equals elementType
     */
    boolean checkArrayType (Object[] o, int elementType);

    /**
     * This function test whether the two objects specified are peers.
     *
     * @param o1
     * @param o2
     * @return true if o1 and o2 share the same owner
     */
    boolean isPeer (Object o1, Object o2);
    /**
     * Tests whether owner is the owner of obj.
     *
     * @param owner
     * @param child
     * @return true if owner is the owner of obj
     */
    boolean isOwner (Object owner, Object obj);

    /**
     * Returns the owner of an object.
     *
     * Might be null if the owner is not in registered to the hashtable.
     *
     * @param obj  the object
     * @return     the owner for obj
     */
    Object getOwner (Object obj);

    /**
     * Get a member of the rootset. To be able to put objects to the rootset
     * (for static initializers ) or do things relative to the rootset (for
     * example in the policy).
     *
     * @return a member of the root set
     */
    Object getRootSetMember ();

    /**
     * Saves the current context in the current threads hashtable entry.
     * This information is later retrieved by getContext to run static
     * functions in the same universe as the function they were called from.
     * ( static functions use getContext() instead of the not existing "this"
     * reference)
     *
     * @param currentObject the current context
     */
    void setContext (Object currentObject);

    /**
     * Resets the saved context to the value it had before the last setContext().
     * ie. pops the top of the stack.
     */
    void resetContext ();

    /**
     * Returns the current context.
     *
     * @return the current context
     */
    Object getContext ();
}
```

Listing C.2: The interface that has to be implemented by alternative implementations.

```
/*
 * This file is part the Universe Runtime Classes.
 *
 * Copyright (C) 2003-2005 Swiss Federal Institute of Technology Zurich
 *
 * Part of mjc, the MultiJava Compiler.
 *
 * Copyright (C) 2000-2005 Iowa State University
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
 *
 * $Id$
 */

package org.multijava.universes.rt;

/**
 * This interface has to be implemented by any class that is intended to
 * implement the policies needed for the Universe Runtime Checks.
 *
 * @author scdaniel
 */
public interface UrtPolicy {
    /**
     * Defines how external objects (from non-universe aware code) are handled.
     * Are they peer?
     *
     * @return true if external objects should be handled as being peer
     */
    boolean isExternalPeer ();

    /**
     * Defines the owner for Universe aware Objects that are created
     * from outside of Universe aware code.
     *
     *  Possibilities are a virtual native-set or the root set or ...
     * If null is returned, the implementation should default to something
     * reasonable (like registering the object peer with regards to the last
     * object).
     *
     * @param obj  the object that needs an owner
     *
     * @return the owner
     */
    Object getNativeOwner (Object obj);

    /**
     * Defines what to do when a reference to the owner is requested and
     * the owner has already been collected.
     *
     * @param obj  the Object whose owner has already been collected and is
     *             now requested.
     *
     * @return the object to use as owner of obj
     */
    Object getCollectedOwner (Object obj);

    /**
     * Handles an illegal cast.
     */
    void illegalCast ();

    /**
     * Handles an illegal array store.
     */
    void illegalArrayStore ();
}
```

Listing C.3: The interface that has to be implemented by alternative policies.

C.1.3 Implementations

```
/*
 * This file is part the Universe Runtime Classes.
 *
 * Copyright (C) 2003-2005 Swiss Federal Institute of Technology Zurich
 *
 * Part of mjc, the MultiJava Compiler.
 *
 * Copyright (C) 2000-2005 Iowa State University
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
 *
 * $Id$
 */

package org.multijava.universes.rt.impl;

import java.lang.ref.ReferenceQueue;

import org.multijava.mjc.MjcTokenTypes;
import org.multijava.universes.rt.UniverseRuntime;
import org.multijava.universes.rt.UrtImplementation;
import org.multijava.util.Utils;

/**
 * This Class manages the dynamic type checking for the universe type
 * system. It uses an optimized IntHashtable (UrtHashtable)
 * to store the ownership relations.
 *
 * Newly created objects should be registered to the hashtable by either
 * calling setConstructorData before the object is created and then calling
 * setOwner in the constructor (before field initializations !!) or by
 * calling setOwnerPeer or setOwnerRep after the object has been created.
 *
 * Whenever there's a universe aware constructor, the first method has to
 * be used in order to make sure objects created in the constructor and
 * field initializations can be registered too.
 *
 * Furthermore it provides functions to check if two objects are peers or
 * if one is the owner of the other.
 * Functions to handle errors are implemented too, so alternative
 * implementations of this class can handle errors differently.
 *
 * Alternative implementations have to implement
 * UrtImplementation.
 *
 * @author scdaniel
 */
public class UrtDefaultImplementation extends Utils implements UrtImplementation {
    // the hashtable used to store the ownership relations
    private UrtHashtable table;
    // the ReferenceQueue that "informs" us about deleted objects so we can cleanup
    private ReferenceQueue queue;

    /**
     * Constructor.
     * Initializes things and sets up the "root" of the ownership relation
     * hierarchy by inserting a "root" element and the current thread into
     * the hashtable.
     * We only need this to be able to handle objects in the root set the
     * same way as normal objects.
     * (Objects in the root set don't have an owner; Examples: the first
     * Thread that is created and runs the program or peer objects created
     * in the static (!) main function.)
     */
    public UrtDefaultImplementation () {
        table = new UrtHashtable();
        queue = new ReferenceQueue();

        /* put the table itself on top of the hierarchy */
        UrtHashtableEntry e = table.put(table, new UrtHashtableEntry());
        e.obj = new UrtWeakReference(table);
        e.owner = null;

        /* add the current thread as the first object to the root-set */
        setOwnerRep(Thread.currentThread(), table);
        setOwnerRep(queue, table);
    }

    /**
     * Registers the ownership relation of obj and owner in the hashtable.
     * (owner is the owner of obj)
     *
     * @param obj the object
     * @param owner it's owner
     */
    public void setOwnerRep (Object obj, Object owner) {
        assertTrue(obj != null);
        assertTrue(owner != null);

        /* try to put the object into the hashtable */
        UrtHashtableEntry e;
        if ( obj instanceof Thread )
            e = setOwner(obj, owner, new UrtHashtableThreadEntry(obj, queue));
        else
            e = setOwner(obj, owner, new UrtHashtableEntry());
    }

    /**
     * Registers the ownership relation of obj and current in the hashtable.
     * (they both share the same owner)
     *
     * @param obj      the object
     * @param current  the owner of the current universe when obj is created
     */
    public void setOwnerPeer (Object obj, Object current) {
        assertTrue(obj != null);
        assertTrue(current != null);

        /* get owner of the current object ... */
        UrtHashtableEntry e = table.get(current);
        assertTrue(e != null);
        // FIXME: potential problem if owner of current object already collected
        setOwnerRep(obj, e.owner.obj.get());
    }
    /**
     * Saves the current object and the modifier for the object being
     * created in its context in the current threads hashtable entry.
     * This information is later retrieved by setOwner (which is called in
     * the constructor) to set the owner of newly created objects before
     * they create any other objects in the constructor.
     *
     * @param currentObject the object that owns the current universe
     * @param objectClass   the class of the object that is to be created
     * @param modifier      the modifier that applies to the object to be created
     *                      (one of MjcTokenTypes.LITERAL_peer, MjcTokenTypes.LITERAL_rep)
     */
    public void setConstructorData (Object currentObject, Object objectClass, int modifier) {
        assertTrue(currentObject != null);
        assertTrue(objectClass != null);

        UrtHashtableThreadEntry e = getCurrentThreadEntry();

        e.currentObject = currentObject;
        e.objectClass = objectClass;
        e.modifier = modifier;
    }

    /**
     * Sets the owner of the current object by using the data stored in the
     * the current threads hashtable entry.
     * This function should be called in the constructor, before any object
     * (including field initializers ) is created.
     *
     * @param o the current object
     */
    public void setOwner (Object o) {
        assertTrue(o != null);

        UrtHashtableThreadEntry e = getCurrentThreadEntry();

        /* there has to be some data associated with this threads entry */
        assertTrue(e.currentObject != null);
        assertTrue(e.objectClass != null);

        /*
         * If the classes do not match, the data has been saved in the threads
         * entry before the creation of a native Java Object (not universe aware)
         * and the Object being created now (universe aware) is created from
         * outside of universe aware code. Therefore the saved date does not
         * apply.
         *
         * Get the owner for this Object and if no special owner is defined,
         * register it as peer relative to the last object.
         */
        if ( e.objectClass != o.getClass() ) {
            Object owner = UniverseRuntime.policy.getNativeOwner(o);
            if ( owner == null )
                setOwnerPeer(o, e.currentObject);
            else
                setOwnerRep(o, owner);

            return;
        }

        /* call the right function */
        switch ( e.modifier ) {
            case MjcTokenTypes.LITERAL_peer: setOwnerPeer(o, e.currentObject); break;
            case MjcTokenTypes.LITERAL_rep: setOwnerRep(o, e.currentObject); break;
            default: assertTrue(e.modifier != e.modifier) ; break;
        }
    }

    /**
     * Registers the ownership relation of the array obj and the object
     * owner in the hashtable. (owner is the owner of obj)
     * And stores the universe type of the array elements in the hashtable
     * entry that is used for the object.
     *
     * @param obj  the object
     * @param owner it's owner
     */
    public void setArrayOwnerRep (Object[] obj, Object owner, int arrayElementType) {
        assertTrue(obj != null);
        assertTrue(owner != null);

        UrtHashtableArrayEntry e = (UrtHashtableArrayEntry)
            setOwner(obj, owner, new UrtHashtableArrayEntry(arrayElementType));
    }

    /**
     * Registers the ownership relation of the array obj and the object
     * current in the hashtable. (they both share the same owner)
     * And stores the universe type of the array elements in the hashtable
     * entry that is used for the object.
     *
     * @param obj       the array
     * @param current  the owner of the current universe when obj is created
     */
    public void setArrayOwnerPeer (Object[] obj, Object current, int arrayElementType) {
        assertTrue(obj != null);
        assertTrue(current != null);

        /* get owner of the current object ... */
        UrtHashtableEntry e = table.get(current);
        assertTrue(e != null);
        // FIXME: potential problem if owner of current object already collected
        setArrayOwnerRep(obj, e.owner.obj.get(), arrayElementType);
    }

    /**
     * Checks if the array specified has exactly the elementType specified.
     *
     * For ArrayStores, we need to know whether it is really readonly.
     * That's why it does not work the same as instanceof and therefor
     * returns only true if the actual element type is equal to elementType.
     *
     * @param o              the array to check
     * @param elementType the element type to check for
     * @return true if the elementType of o equals elementType
     */
    public boolean checkArrayType (Object[] o, int elementType) {
        /* null objects don't have elementTypes */
        if ( o == null )
            return false;

        UrtHashtableArrayEntry e =
            (UrtHashtableArrayEntry) table.get(o);

        /*
         * if the object is not in the hashtable, handle external objects
         * as defined in isExternalPeer() if elementType is peer.
         * Otherwise if element type is rep return false and if it is
         * readonly return true.
         */
        if ( e == null )
            if ( elementType == MjcTokenTypes.LITERAL_peer )
                return UniverseRuntime.policy.isExternalPeer();
            else if ( elementType == MjcTokenTypes.LITERAL_rep )
                return false;
            else
                return true;

        return (e.elementType == elementType);
    }

    /**
     * This function test whether the two objects specified are peers.
     *
     * @param o1
     * @param o2
     * @return true if o1 and o2 share the same owner
     */
    public boolean isPeer (Object o1, Object o2) {
        /* null objects don't have the same parent as anything else */
        if ( (o1 == null) || (o2 == null) )
            return false;

        UrtHashtableEntry e1 = getOwnerEntry(o1);
        UrtHashtableEntry e2 = getOwnerEntry(o2);

        /*
         * was one of the objects created outside of universe aware code?
         */
        boolean result;
        if ( (e1 == null) || (e2 == null) )
            return UniverseRuntime.policy.isExternalPeer();

        return (e1 == e2);
    }

    /**
     * Tests whether owner is the owner of obj.
     *
     * @param owner
     * @param child
     * @return true if owner is the owner of obj
     */
    public boolean isOwner (Object owner, Object obj) {
        /* null objects don't have or are owners */
        if ( (owner == null) || (obj == null) )
            return false;

        UrtHashtableEntry r1 = table.get(owner);
        UrtHashtableEntry r2 = getOwnerEntry(obj);

        /*
         * if the object pretending to be child was created outside of universe
         * aware code, so owner can't be it's owner.
         * The same holds if the owner was not found.
         */
        if ( (r1 == null) || (r2 == null) )
            return false;

        return (r1 == r2);
    }
    /**
     * Returns the owner of an object.
     *
     * Might be null if the owner is not in registered to the hashtable.
     *
     * @param obj  the object
     * @return      the owner for obj
     */
    public Object getOwner (Object obj) {
        assertTrue(obj != null);

        UrtHashtableEntry ownerEntry = getOwnerEntry(obj);

        if ( ownerEntry == null )
            return null;

        /* what to do if the owner field is null? */
        if ( ownerEntry.obj == null )
            return UniverseRuntime.policy.getCollectedOwner(obj);

        return ownerEntry.obj.get();
    }

    /**
     * Get a member of the rootset. To be able to put objects to the rootset
     * (for static initializers ) or do things relative to the rootset (for
     * example in the policy).
     *
     * @return a member of the root set
     */
    public Object getRootSetMember () {
        return queue;
    }

    /**
     * Saves the current context in the current threads hashtable entry.
     * This information is later retrieved by getContext to run static
     * functions in the same universe as the function they were called from.
     * ( static functions use getContext() instead of the not existing "this"
     * reference)
     *
     * @param currentObject the current context
     */
    public void setContext (Object currentObject) {
        assertTrue(currentObject != null);

        UrtHashtableThreadEntry e = getCurrentThreadEntry();

        e.push(currentObject);
    }

    /**
     * Resets the saved context to the value it had before the last setContext().
     * ie. pops the top of the stack.
     */
    public void resetContext () {
        UrtHashtableThreadEntry e = getCurrentThreadEntry();

        e.pop();
    }

    /**
     * Returns the current context.
     *
     * @return the current context
     */
    public Object getContext () {
        UrtHashtableThreadEntry e = getCurrentThreadEntry();

        return e.getCurrentContext();
    }


    /* ----------- PRIVATE FUNCTIONS ----------- */

    /**
     * Internal function that does the actual insert into the hashtable.
     * Avoids code duplication.
     *
     * To allow different object types to have more data associated with
     * them, the hashtable entry that should be used for the object has
     * to be given. (ie. Arrays store the elementType, ...)
     *
     * @param obj       the object to insert
     * @param owner     the owner of that object
     * @param e         the hashtable entry to use for that object
     *
     * @return the hashtable entry of the object
     */
    private UrtHashtableEntry setOwner (Object obj, Object owner,
            UrtHashtableEntry e) {
        /* try to put the object into the hashtable */
        e = table.put(obj, e);

        /* if it was already registered , exit */
        if ( (e.owner != null) )
            return e;

        /*
         *  Create a weak reference to the object and register
         *  with the reference queue. To be noticed, when the
         *  object is collected .
         */
        e.obj = new UrtWeakReference(obj, queue);

        /* Set owner */
        e.owner = table.get(owner);
        /* we need to have an owner here */
        assertTrue(e.owner != null);

        e.owner.children++;

        /*
         * Reset the threads saved data, so constructors or functions called
         * from non-universe aware code are handled correctly.
         */
        UrtHashtableThreadEntry thread = getCurrentThreadEntry();
        thread.currentObject = owner;
        thread.objectClass = Object.class;
        thread.modifier = MjcTokenTypes.LITERAL_peer;

        /*
         *  Check for collected objects and remove unnecessary
         *  stuff in our data structure to make it ready to be
         *  collected too.
         */
        UrtWeakReference o;
        UrtHashtableEntry i;
        while ( (o = (UrtWeakReference) queue.poll()) != null ) {
            i = (UrtHashtableEntry) table.get(o);
            /* clear WeakLink so the Object can be collected */
            i.obj = null;
            /* the second test is for further iterations */
            while ( (i.children == 0) && (i.obj == null) ) {
                table.remove(i);
                i.owner.children--;

                /*
                 *  If the owner has no more children now, and
                 *  its object already has been collected , we
                 *  may delete this one too.
                 */
                i = i.owner;

                if ( i == null )
                    break;
            }
        }

        return e;
    }

    /**
     * Internal function that gets the hashtable entry of an objects owner.
     *
     * @param obj  the object
     * @return     the hashtable entry of that objects owner
     */
    private UrtHashtableEntry getOwnerEntry (Object obj) {
        UrtHashtableEntry e = table.get(obj);

        /*
         *  every object should be in our table
         *  if not, it has been created outside of universe aware code
         */
        if ( e == null )
            return null;

        /* every object should have an owner */
        assertTrue(e.owner != null);

        return e.owner;
    }

    /**
     * Get the current Threads HashtableEntry and create it if it does not exist.
     * (It may not exist if the Thread has been created outside of universe
     * aware code)
     *
     * @return the current Threads UrtHashtableThreadEntry
     */
    private UrtHashtableThreadEntry getCurrentThreadEntry () {
        UrtHashtableThreadEntry thread =
            (UrtHashtableThreadEntry) table.get(Thread.currentThread());
        if ( thread == null ) {
            setOwnerRep(Thread.currentThread(), table);
            thread =
                (UrtHashtableThreadEntry) table.get(Thread.currentThread());
        }

        return thread;
    }
}
```

Listing C.4: The default implementation.

```
/**
 * UrtHashtable - a Hashtable that uses ints as the keys
 *
 * based on IntHashtable by the ACME Labs:
 * <a href="http://www.acme.com/java/software/Acme.IntHashtable.html">ACME IntHashtable</a>
 *
 * This one is highly optimized to be used for the dynamic checks
 * of the universe type system. (@see UniverseRuntime)
 *
 * scdaniel - Changelog
 * ---------------------
 * - removed:
 *     enumeration
 *     all but the default constructor
 *     size()
 *     isEmpty()
 *     contains()
 *     containsKey()
 *     put(Object, Object)
 *     get(Object)
 *     remove(Object)
 *     toString()
 *     clone()
 * - changes to get(), put() and remove()
 *   details are found in the respective javadocs
 */

package org.multijava.universes.rt.impl;

import java.lang.ref.ReferenceQueue;
import java.lang.ref.WeakReference;

import org.multijava.mjc.MjcTokenTypes;
import org.multijava.util.Utils;

public class UrtHashtable {
    // The hash table data.
    private UrtHashtableEntry table[];
    // The total number of entries in the hash table.
    private int count;
    // Rehashes the table when count exceeds this threshold.
    private int threshold;
    // The load factor for the hashtable.
    private float loadFactor;

    // Constructs a new, empty hashtable. A default capacity and load factor
    // is used. Note that the hashtable will automatically grow when it gets
    // full.
    public UrtHashtable() {
        int initialCapacity = 101;
        float loadFactor = 0.75f;

        if ( initialCapacity <= 0 || loadFactor <= 0.0 )
            throw new IllegalArgumentException();
        this.loadFactor = loadFactor;
        table = new UrtHashtableEntry[initialCapacity];
        threshold = (int) (initialCapacity * loadFactor);
    }

    /**
     * Gets the record associated with the specified object.
     * System.identityHashCode() is used as key.
     *
     * @param o the object to search for
     *
     * @return the element for the object or null if the
     *         object is not defined in the hash table.
     *
     *
     * scaniel - Changelog
     * ------------------
     * - use an Object instead of an int key to get the record
     * - return the UrtHashtableEntry instead of a value
     * - removed tab, which was a reference to table, for simplicity
     *
     * This is needed, since we need to make sure it is the
     * right Entry, ie. the one linking to the Object.
     * And it works, since the int key is System.identitHashCode
     * of the Object.
     */
    public synchronized UrtHashtableEntry get (Object o) {
        // scdaniel
        int hash = System.identityHashCode(o);

        int index = (hash & 0x7FFFFFFF) % table.length;
        for ( UrtHashtableEntry e = table[index]; e != null; e = e.next ) {
            // scdaniel: last test needed to make sure it's the right one
            if ( e.hash == hash && e.holds(o) )
                return e;
        }
        return null;
    }

    /**
     * Gets the record that uses the specified UrtWeakReference
     * to link to the object it is associated with.
     *
     * @param r the UrtWeakReference to search for
     *
     * @return the element for the object or null if the
     *         object is not defined in the hash table.
     */
    public synchronized UrtHashtableEntry get (UrtWeakReference r) {
        int hash = r.hash;

        int index = (hash & 0x7FFFFFFF) % table.length;
        for ( UrtHashtableEntry e = table[index]; e != null; e = e.next ) {
            if ( e.hash == hash && e.obj == r )
                return e;
        }
        return null;
    }

    // Rehashes the content of the table into a bigger table.
    // This method is called automatically when the hashtable's
    // size exceeds the threshold.
    protected void rehash () {
        int oldCapacity = table.length;
        UrtHashtableEntry oldTable[] = table;
        int newCapacity = oldCapacity * 2 + 1;
        UrtHashtableEntry newTable[] = new UrtHashtableEntry[newCapacity];
        threshold = (int) (newCapacity * loadFactor);
        table = newTable;
        for ( int i = oldCapacity; i-- > 0; ) {
            for ( UrtHashtableEntry old = oldTable[i]; old != null; ) {
                UrtHashtableEntry e = old;
                old = old.next;
                int index = (e.hash & 0x7FFFFFFF) % newCapacity;
                e.next = newTable[index];
                newTable[index] = e;
            }
        }
    }

    /**
     * Puts the specified element into the hashtable, using
     * System.identityHashCode() as key.
     * The element and the entry cannot be null.
     *
     * @param o the element to add
     * @param e the entry to use for this element
     *
     * @return the entry in the hashtable for this object
     *
     * @exception NullPointerException If the object or the
     *            entry is equal to null.
     *
     * scdaniel - Changelog
     * --------------------
     * - use the an Object and an UrtHashtableEntry instead
     *   of a key to allow different entry types
     *   (see IntHashtableThreadEntry)
     * - test if we need to rehash earlier, to save some instructions
     * - also save objects with duplicate hashcodes
     * - removed tab, which was a reference to table, for simplicity
     */
    public synchronized UrtHashtableEntry put (Object o,
            UrtHashtableEntry e) {
        // Make sure the entry is not null.
        if ( (o == null) || (e == null) )
            throw new NullPointerException();

        // scdaniel: do this before the loop, saves some time
        if ( count >= threshold ) {
            // Rehash the table if the threshold is exceeded.
            rehash();
        }

        // scdaniel
        int hash = System.identityHashCode(o);

        int index = (hash & 0x7FFFFFFF) % table.length;
        for ( UrtHashtableEntry i = table[index]; i != null; i = i.next ) {
            /*
             * scdaniel: we want to be able to have objects
             * with duplicate hashCodes, so only return on
             * really duplicate objects
             */
            if ( (i.hash == hash) && i.holds(o) )
                return i;
        }

        // Creates the new entry.
        e.hash = hash;
        e.next = table[index];
        table[index] = e;
        ++count;
        return e;
    }

    /**
     * Removes the specified element.
     * Does nothing if the key is not present.
     *
     * @param e the entry that needs to be removed
     *
     * scdaniel - Changelog
     * --------------------
     * - use the entry itself instead of a key
     * - remove only when the very same (==) element is found
     * - removed tab, which was a reference to table, for simplicity
     */
    public synchronized void remove (UrtHashtableEntry e) {
        int hash = e.hash;
        int index = (hash & 0x7FFFFFFF) % table.length;
        for ( UrtHashtableEntry i = table[index], prev = null;
                i != null; prev = i, i = i.next ) {
            if ( i == e) {
                if ( prev != null )
                    prev.next = i.next;
                else
                    table[index] = i.next;
                --count;
            }
        }
    }
}


/**
 * The class for the entries used in UrtHashtable.
 *
 * scdaniel - Changelog
 * --------------------
 * - removed value, we're using the entry itself to store everything
 * - removed key, it's equal to hash
 * - added some attributes used for the ownerHashtable
 *
 * The fields are accessible directly for speed reasons.
 */
class UrtHashtableEntry {
    int hash;
    UrtHashtableEntry next;

    /*
     * scdaniel
     *
     * We need the following attributes for our ownerHashtable.
     */
    UrtWeakReference obj;
    UrtHashtableEntry owner;
    int children = 0;

    /**
     * Tests if this instance of UrtHashtableEntry holds
     * the Object specified. (ie. links to it)
     *
     * @param o  the Object to test
     * @return true if this UrtHashtableEntry holds the specified Object
     */
    public boolean holds (Object o) {
        return ( this.obj.get() == o );
    }
}

/**
```

```
 * The class used for entries associated to arrays. Stores the element type
 * in addition to the normal things.
 *
 * The fields are accessible directly for speed reasons.
 */
class UrtHashtableArrayEntry extends UrtHashtableEntry {
    int elementType;

    /**
     * Constructor that sets the element type of the array this entry is
     * representing.
     *
     * @param elementType the element type
     *                        (one of MjcTokenTypes.LITERAL_peer, MjcTokenTypes.LITERAL_rep)
     */
    public UrtHashtableArrayEntry (int elementType) {
        this.elementType = elementType;
    }
}

/**
 * This class is used for entries associated to threads.
 * Provides additional information to allow to set the owner in the constructor
 * and therefore associate objects created in the constructor (and fields) with
 * their correct owners.
 * This wouldn't work when setting the owner after the "new" operator.
 *
 * It also saves the context history needed for static functions.
 * Before a static function is called from a non-static context, the current
 * object is put on top of the stack so inside the static function, the same
 * context can be used. When the static function returns, the top element of
 * the stack is removed again.
 *
 * The fields are accessible directly for speed reasons.
 */
class UrtHashtableThreadEntry extends UrtHashtableEntry {
    Object currentObject;
    int modifier;
    Object objectClass;
    private UrtStackItem contexts;

    /**
     * Constructor that takes the object it will link to as argument to
     * initialize the additional data in case this thread is used to
     * create a universe aware object while it is in a non-universe aware
     * context.
     *
     * @param currentObject the object it is to be associated with
     * @param root the root object of the context stack
     */
    public UrtHashtableThreadEntry (Object currentObject, Object root) {
        this.currentObject = currentObject;
        modifier = MjcTokenTypes.LITERAL_peer;
        this.objectClass = null;
        contexts = new UrtStackItem(root, null);
    }

    /**
     * Push the current context on the stack.
     *
     * @param context
     */
    public synchronized void push (Object context) {
        contexts = new UrtStackItem(context, contexts);
    }

    /**
     * Remove the topmost stack element.
     */
    public synchronized void pop () {
        Utils.assertTrue(contexts != null);

        contexts = contexts.next;
    }

    /**
     * Get the current context. Without changing the stack.
     *
     * @return the topmost stack element
     */
    public Object getCurrentContext () {
        Utils.assertTrue(contexts != null);

        return contexts.obj;
    }
}

/**
 * This is a simple stackItem that just stores a reference to an object and
 * a reference to the next item on the stack.
 *
 * The fields are accessible directly for speed reasons.
 */
class UrtStackItem {
    Object obj;
    UrtStackItem next;

    UrtStackItem (Object obj, UrtStackItem next) {
        this.obj = obj;
        this.next = next;
    }
}

/**
 * A class that extends WeakReference and adds the additional attribute
 * hash storing the System.identityHashCode of the object it referes to.
 * This allows to delete the hashtable entry of that object if the object
 * is collected and we are noticed by the reference queue.
 *
 * The fields are accessible directly for speed reasons.
 */
class UrtWeakReference extends WeakReference {
    // the hashcode of the object we're linking to
    int hash;

    // just calls super and sets the hashcode
    UrtWeakReference (Object o) {
        super(o);
        hash = System.identityHashCode(o);
    }

    // just calls super and sets the hashcode
    UrtWeakReference (Object o, ReferenceQueue q) {
        super(o, q);
        hash = System.identityHashCode(o);
    }
}
```

Listing C.5: The modified IntHashtable used by the default implementation (see listing C.4).

```
/*
 * This file is part the Universe Runtime Classes.
 *
 * Copyright (C) 2003-2005 Swiss Federal Institute of Technology Zurich
 *
 * Part of mjc, the MultiJava Compiler.
 *
 * Copyright (C) 2000-2005 Iowa State University
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
 *
 * $Id$
 */

package org.multijava.universes.rt.impl;

import org.multijava.universes.rt.UrtImplementation;

/**
 * Alternative implementation of UrtDefaultImplementation that prints debug
 * information to stdout before it calls the respective method of
 * UrtDefaultImplementation.
 *
 * @author scdaniel
 */
public class UrtDebug
        extends UrtDefaultImplementation
        implements UrtImplementation {
    public void setOwnerRep (Object obj, Object owner) {
        System.out.println("setOwnerRep: " + obj + ", " + owner);
        super.setOwnerRep(obj, owner);
    }

    public void setOwnerPeer (Object obj, Object current) {
        System.out.println("setOwnerPeer: " + obj + ", " + current);
        super.setOwnerPeer(obj, current);
    }

    public void setConstructorData (Object currentObject, Object objectClass, int modifier) {
        System.out.println("setConstructorData: " + currentObject + ", " +
                objectClass + ", " + modifier);
        super.setConstructorData(currentObject, objectClass, modifier);
    }

    public void setOwner (Object o) {
        System.out.println("setOwner: " + o);
        super.setOwner(o);
    }

    public void setArrayOwnerRep (Object[] obj, Object owner, int arrayElementType) {
        System.out.println("setArrayOwnerRep: " + obj + ", " + owner + ", " +
                arrayElementType);
```

```java
        super.setArrayOwnerRep(obj, owner, arrayElementType);
    }

    public void setArrayOwnerPeer (Object[] obj, Object current,
            int arrayElementType) {
        System.out.println("setArrayOwnerPeer: " + obj + ", " + current + ", " +
                arrayElementType);
        super.setArrayOwnerPeer(obj, current, arrayElementType);
    }

    public boolean isPeer (Object o1, Object o2) {
        System.out.println("isPeer: " + o1 + ", " + o2);
        return super.isPeer(o1, o2);
    }

    public boolean isOwner (Object owner, Object obj) {
        System.out.println("isOwner: " + owner + ", " + obj);
        return super.isOwner(owner, obj);
    }

    public boolean checkArrayType (Object[] o, int elementType) {
        System.out.println("checkArrayType: " + o + ", " + elementType);
        return super.checkArrayType(o, elementType);
    }

    public Object getOwner (Object obj) {
        System.out.println("getOwner: " + obj);
        return super.getOwner(obj);
    }

    public Object getRootSetMember () {
        System.out.println("getRootSetMember");
        return super.getRootSetMember();
    }

    public void setContext (Object currentObject) {
        System.out.println("setContext: " + currentObject);
        super.setContext(currentObject);
    }

    public void resetContext () {
        System.out.println("resetContext");
        super.resetContext();
    }

    public Object getContext () {
        System.out.println("getContext");
        return super.getContext();
    }
}
```

Listing C.6: `UrtDebug.java`

```java
/*
 * This file is part the Universe Runtime Classes.
 *
 * Copyright (C) 2003-2005 Swiss Federal Institute of Technology Zurich
 *
 * Part of mjc, the MultiJava Compiler.
 *
 * Copyright (C) 2000-2005 Iowa State University
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
 *
 * $Id$
 */

package org.multijava.universes.rt.impl;

import org.multijava.universes.rt.UrtImplementation;

/**
 * Alternative implementation of UrtDefaultImplementation that does nothing.
 *
 * Usefull to speed up universe programs without having to recompile it without
 * universe runtime check support.
 *
 * @author scdaniel
 */
public class UrtDummy implements UrtImplementation {

    public void setOwnerRep (Object obj, Object owner) {}
    public void setOwnerPeer (Object obj, Object current) {}
    public void setConstructorData (Object currentObject, Object objectClass, int modifier) {}
    public void setOwner (Object o) {}
    public void setArrayOwnerRep (Object[] obj, Object owner, int arrayElementType) {}
    public void setArrayOwnerPeer (Object[] obj, Object current, int arrayElementType) {}
    public boolean isPeer (Object o1, Object o2) { return true; }
    public boolean isOwner (Object owner, Object obj) { return true; }
    public boolean checkArrayType (Object[] o, int elementType) { return true; }
    public Object getOwner (Object obj) { return null; }
    public Object getRootSetMember () { return null; }
    public void setContext (Object currentObject) {}
    public void resetContext () {}
    public Object getContext () { return null; }
}
```

Listing C.7: `UrtDummy.java`

```
/*
 * This file is part the Universe Runtime Classes.
 *
 * Copyright (C) 2003-2005 Swiss Federal Institute of Technology Zurich
 *
 * Part of mjc, the MultiJava Compiler.
 *
 * Copyright (C) 2000-2005 Iowa State University
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
 *
 * $Id$
 */

package org.multijava.universes.rt.impl;

import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;

import org.multijava.universes.rt.UrtImplementation;

/**
 * Alternative implementation of UrtDefaultImplementation that generates
 * dot code to generate a visualization of the ownership relations.
 *
 * It prints the code to the file specified in the Java property "UrtOutfile".
 * The language used to describe the graph is "dot", which is part of the
 * graphviz package.
 * @see http://www.graphviz.org/
 *
 * The name of the objects can be modified by changing the getName() function.
 * Standard is "uniqueNumber - shortClassName@identityHashcode".
 * The unique number is a counter lets you see the order in which objects were
 * created.
 *
 * To automate this whole thing just put the something like the following lines
 * in your Makefile:
 *
 *  visualize: compile
 *       java -DUniverseRuntime=org.multijava.universes.rt.UrtVisualizer \
 *            -DUrtOutfile=dot.dot helloWorld
 *       dot -o dot.ps -Tps dot.dot
 *       gv dot.ps &
 *
 * @author scdaniel
 */
public class UrtVisualizer
        extends UrtDefaultImplementation
        implements UrtImplementation {
```

```java
    /**
     * Objects needed to print to a file .
     */
    FileWriter fw;
    PrintWriter pw;
    // the counter
    int counter = 0;

    /**
     * Opens the outfile and prints the information that has already been
     * collected by functions called in the constructor of the superclasses .
     *
     * @throws IOException
     */
    public UrtVisualizer () throws IOException {
        /* open file and start graph */
        String outfile = System.getProperty("UrtOutfile");
        if ( outfile == null )
            throw new IOException(UrtVisualizer.class.toString() + ": no outputfile defined!");
        fw = new FileWriter(new File( outfile ));
        pw = new PrintWriter(fw);
        pw.println("digraph UniverseRuntime {");

        Runtime.getRuntime().addShutdownHook(new Thread() {
                public void run() {
                    cleanup();
                }
            });
    }

    /**
     * Generates a name for an object.
     *
     * @param o the object
     * @return the name
     */
    public String getName (Object o) {
        String hashCode = Integer.toString(System.identityHashCode(o));
        String clazz = o.getClass().getName();
        int index = clazz.lastIndexOf(".");
        if ( index == -1 )
            return clazz + "@" + hashCode;
        else
            return clazz.substring(index + 1) + "@" + hashCode;
    }

    public void setOwnerRep (Object obj, Object owner) {
        /*
         * If pw is initialized , everything is fine , if not, we have to save the
         * data elsewhere until the constructor is called .
         */
        if ( pw != null ) {
            String ownerName = getName(owner);
            String objName = getName(obj);
            /*
             * Don't display the head of the structure (the "virtual" owner of
             * the root set).
             */
            if ( !( owner instanceof UrtHashtable) )
                pw.println("\t\"" + ownerName + "\" -> \"" + objName + "\";");
            pw.println("\t\"" + objName + "\"[label=\"" + (counter++) + " - " + objName + "
                \"];");
            pw.println("\tsubgraph \"cluster" + ownerName + "\" {\"" + objName + "\";}");
            pw.flush();
        } else {
            /*
             * The objects registered in the superclass constructor don't need
             * to be displayed. It's just the Thread and the Queue both of which
             * should never have objects in their universe.
             * If they have it's a bug. :)
             * And it will be shown nevertheless (when the objects that reference
             * the thread or the queue as owner are added).
             */
        }
        super.setOwnerRep(obj, owner);
    }

    /**
     * Finish the graph and close the file.
     */
    private void cleanup () {
        pw.println("}");
        pw.flush();
        try {
            fw.close();
        } catch ( IOException e ) {}
    }
}
```

Listing C.8: `UrtVisualizer.java`

C.1.4 Policies

```
/*
 * This file is part the Universe Runtime Classes.
 *
 * Copyright (C) 2003-2005 Swiss Federal Institute of Technology Zurich
 *
 * Part of mjc, the MultiJava Compiler.
 *
 * Copyright (C) 2000-2005 Iowa State University
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
 *
 * $Id$
 */

package org.multijava.universes.rt.policy;

import org.multijava.universes.rt.UrtPolicy;
import org.multijava.util.Utils;

/**
 * This class defines how to handle errors and defines a few further policies.
 *
 * Alternative implementations have to implement UrtPolicy.
 *
 * @author scdaniel
 */
public class UrtDefaultPolicy extends Utils implements UrtPolicy {
    /**
     * Defines how external objects (from non-universe aware code) are
     * handled. i.e.: "Are they peer?"
     *
     * @return true if external objects should be handled as being peer
     */
    public boolean isExternalPeer () {
        return true;
    }

    /**
     * Universe objects generated from external code should be registered
     * as peer relative to the last object.
     *
     * @return null
     */
    public Object getNativeOwner (Object obj) {
        return null;
    }

    /**
     * If somebody asks for an already collected owner, return null.
     *
     * @return null
     */
    public Object getCollectedOwner (Object obj) {
        return null;
    }

    /**
     * Handles an illegal cast by throwing a ClassCastException.
     */
    public void illegalCast () {
        throw new ClassCastException();
    }

    /**
     * Handles an illegal array store exception by throwing an
     * ArrayStoreException.
     */
    public void illegalArrayStore () {
        throw new ArrayStoreException();
    }
}
```

Listing C.9: `UrtDefaultPolicy.java`

```
/*
 * This file is part the Universe Runtime Classes.
 *
 * Copyright (C) 2003-2005 Swiss Federal Institute of Technology Zurich
 *
 * Part of mjc, the MultiJava Compiler.
 *
 * Copyright (C) 2000-2005 Iowa State University
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
 *
 * $Id$
 */

package org.multijava.universes.rt.policy;

import org.multijava.universes.rt.UrtPolicy;

/**
 * Alternative implementation of UrtDefaultPolicy that does nothing.
 *
 * Usefull to speed up universe programs without having to recompile it without
 * universe runtime check support.
 *
 * @author scdaniel
 */
public class UrtDummy implements UrtPolicy {
    public boolean isExternalPeer () { return true; }
    public Object getNativeOwner (Object obj) { return null; }
    public Object getCollectedOwner (Object obj) { return null; }
    public void illegalCast () {}
    public void illegalArrayStore () {}
}
```

Listing C.10: `UrtDummy.java`

```
/*
 * This file is part the Universe Runtime Classes.
 *
 * Copyright (C) 2003-2005 Swiss Federal Institute of Technology Zurich
 *
 * Part of mjc, the MultiJava Compiler.
 *
 * Copyright (C) 2000-2005 Iowa State University
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
 *
 * $Id$
 */

package org.multijava.universes.rt.policy;

import org.multijava.universes.rt.UrtPolicy;

/**
 * Alternative implementation of UrtDefaultPolicy that does not throw an
 * error on casts from readonly objects that are illegal but just print a warning.
 *
 * @author scdaniel
 */
public class UrtRelaxed
        extends UrtDefaultPolicy
        implements UrtPolicy {
    /**
     * Don't throw an error but just print a message.
     */
    public void illegalCast () {
        try {
            throw new ClassCastException();
        } catch ( ClassCastException e ) {
            System.err.println("Universe Warning: there has been an illegal cast from
                 readonly!");
            printException(e);
        }
    }

    /**
     * Don't throw an error but just print a message.
     */
    public void illegalArrayStore () {
        try {
            throw new ArrayStoreException();
        } catch ( ArrayStoreException e ) {
            System.err.println("Universe Warning: there has been an illegal write access to
                 an array element!");
            printException(e);
        }
    }
    /**
     * Print stack trace without the current function.
     *
     * @param e the exception
     */
    private void printException (Exception e) {
        StackTraceElement[] s = e.getStackTrace();
        for ( int i = 1; i < s.length; i++ )
            System.err.println("    at " + s[i]);
    }
}
```

Listing C.11: `UniverseRuntimeRelaxed.java`

```
/*
 * This file is part the Universe Runtime Classes.
 *
 * Copyright (C) 2003−2005 Swiss Federal Institute of Technology Zurich
 *
 * Part of mjc, the MultiJava Compiler.
 *
 * Copyright (C) 2000−2005 Iowa State University
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111−1307 USA
 *
 * $Id$
 */

package org.multijava.universes.rt.policy;

import org.multijava.universes.rt.UniverseRuntime;
import org.multijava.universes.rt.UrtPolicy;

/**
 * Alternative implementation of UrtDefaultPolicy that does treat objects
 * that have been created in non−universe aware code as readonly and
 * throws an error when somebody asks for an already collected owner.
 *
 * @author scdaniel
 */
public class UrtStrict
        extends UrtDefaultPolicy
        implements UrtPolicy {
    private Object externalOwner;

    /**
     * Constructor.
     * Establishes externalOwner as the owner of a designated special Universe
     * for Universe aware Objects created outside of universe aware code.
     */
    public UrtStrict () {
        externalOwner = new Object();
        UniverseRuntime.handler.setOwnerRep(externalOwner, Thread.currentThread());
    }

    /**
     * External objects are not peer, but readonly.
     *
     * @return false
     */
    public boolean isExternalPeer () {
        return false;
    }

    /**
     * Universe objects generated from external code belong to a separate Universe.
     *
     * @return the owner of the separate "external" Universe
     */
    public Object getNativeOwner (Object obj) {
        return externalOwner;
    }

    /**
     * If somebody asks for an already collected owner, throw a RuntimeException.
     */
    public Object getCollectedOwner () {
        throw new RuntimeException();
    }
}
```

Listing C.12: `UrtStrict.java`

C.2 Benchmarks

```
all : clean compile_compiler
    for i in ` ls *test.java `; do \
        p="`echo $$i | sed -e 's/.java//'`"; \
        echo "Compiling $$i without dynamic checks ..."; \
        java org.multijava.mjc.Main -source 1.4 -universesx parse,check $$i; \
        echo "Running $$i ..."; \
        echo "Profiling $$i ..."; \
        java -Xmx200M -Xrunjmp:nogui,nomethods,dodump,dumpfile=$$p"_nodyn_dump" $$p; \
        echo "Timing $$i ..."; \
        bash -c "time java -Xmx200M $$p" 2>&1 | tee -a $$p"_nodyn_time.txt"; \
        bash -c "time java -Xmx200M $$p" 2>&1 | tee -a $$p"_nodyn_time.txt"; \
        bash -c "time java -Xmx200M $$p" 2>&1 | tee -a $$p"_nodyn_time.txt"; \
        rm -f *.class; \
        echo "Compiling $$i with dynamic checks ..."; \
        java org.multijava.mjc.Main -source 1.4 -universes $$i; \
        echo "Running $$i ..."; \
        echo "Profiling $$i ..."; \
        java -Xmx200M -Xrunjmp:nogui,nomethods,dodump,dumpfile=$$p"_dyn_dump" $$p; \
        echo "Timing $$i ..."; \
        bash -c "time java -Xmx200M $$p" 2>&1 | tee -a $$p"_dyn_time.txt"; \
        bash -c "time java -Xmx200M $$p" 2>&1 | tee -a $$p"_dyn_time.txt"; \
        bash -c "time java -Xmx200M $$p" 2>&1 | tee -a $$p"_dyn_time.txt"; \
    done

compile_compiler:
    make -C ../MultiJava/MJ/org/multijava/mjc clean-this
    make -C ../MultiJava/MJ/org/multijava/mjc all

time:
    for i in ` ls *test.java `; do \
        p="`echo $$i | sed -e 's/.java//'`"; \
        echo "$$i without dynamic checks: "; \
        cat $$p"_nodyn_time.txt" | grep real | sed -e "s/real\t*\([0-9]*\)m\(.*\)s
            /\1*60 + \2 + /" | xargs echo | sed -e "s/\(.*\) +/(\1)\/3/" | bc -l; \
        echo "$$i with dynamic checks: "; \
        cat $$p"_dyn_time.txt" | grep real | sed -e "s/real\t*\([0-9]*\)m\(.*\)s/\1*60 + \2 + /
            " | xargs echo | sed -e "s/\(.*\) +/(\1)\/3/" | bc -l; \
    done

clean:
    rm -f *.class
    rm -f *.txt
```

Listing C.13: The Makefile used to run the tests.

```
public class objecttest {
    public static void main(String[] args) {
        peer peer Object [] a = new peer peer Object[1000000];

        for ( int i = 0; i < 1000000; i++ ) {
            a[i] = new peer Object();
        }
    }
}
```

Listing C.14: `objecttest.java`

```
public class stringtest {
    public static char[] letters = new String("abcdefghijklmnopqrstuvwxyz").toCharArray();

    public static void main(String[] args) {
        Object [] a = new peer peer String[100000];
        char[] c = new char[256];
        Random random = new Random();

        for ( int i = 0; i < 100000; i++ ) {
            for ( int j = 0; j < 256; j ++ ) {
                c[j] = letters [Math.abs(random.nextInt()) % letters.length];
            }
            a[i] = new String(c);
        }
    }
}
```

Listing C.15: `stringtest.java`

```
import java.util.Random;

class listtest {
    public static void main (String[] args) {
        Random random = new Random();
        list l = new list();
        int key;

        for ( int i = 0; i < 1000000; i++ ) {
            key = Math.abs(random.nextInt()) % 10000;
            if ( l.find(key) == null )
                l.insert(key, new Integer(i));
            else {
                if ( (i % 2) == 0 )
                    l.remove(key);
                else
                    l.update(l.find(key), new Integer(i));
            }
        }
    }
}
```

Listing C.16: `listtest.java`

```
class list {
    rep item head;
    rep item tail;

    public readonly item find (int key) {
        rep item i = head;
        while ( i != null ) {
            if ( i.key < key )
                i = i.next;
            else if ( i.key == key )
                return i;
            else
                return null;
        }

        return null;
    }

    public void insert (int key, readonly Object data) {
        rep item i = head;
        while ( i != null ) {
            if ( i.key < key )
                i = i.next;
            else {
                rep item newItem = new rep item(key, data);
                newItem.next = i;
                newItem.prev = i.prev;
                i.prev = newItem;

                if ( newItem.prev == null )
                    head = newItem;
                else
                    newItem.prev.next = newItem;

                return;
            }
        }

        if ( (head == null) || (tail == null) )
            tail = head = new rep item(key, data);
        else {
            rep item newItem = new rep item(key, data);
            tail .next = newItem;
            newItem.prev = tail;
            tail = newItem;
        }
    }

    public void update (readonly item i, readonly Object data) {
        rep item j = (rep item) i;

        j.data = data;
    }

    public void remove (int key) {
        rep item i = head;
        while ( i != null ) {
            if ( i.key < key )
                i = i.next;
            else if ( i.key == key ) {
                if ( i.prev != null )
                    i.prev.next = i.next;
                else
                    head = i.next;
                if ( i.next != null )
                    i.next.prev = i.prev;
                else
                    tail = i.prev;

                return;
            } else
                return;
        }

        return;
    }
}
```

Listing C.17: `list.java`

```
class item {
    int key;

    peer item next;
    peer item prev;

    readonly Object data;

    public item (int key, readonly Object data) {
        this.key = key;
        this.data = data;

        next = null;
        prev = null;
    }
}
```

Listing C.18: `item.java`

C.3 Code comparison

```
class test {
    // static field with initializer
    static peer Object sf = new peer Object();

    // field with initializer
    peer Object f = new peer Object();

    /* static initializer */
    static {
        peer Object o = new peer Object();
    }

    /* constructor */
    public test() {
        f = new peer Object();
    }

    public static void main (String [] args) {
        new peer test().go();
    }

    public void go () {
        /* ------ objects ------ */
        /* new */
        readonly Object o = new peer Object();
        o = new rep Object();

        /* new universe aware object */
        peer test t = new peer test();

        /* instanceof */
        boolean b = o instanceof peer Object;
        b = o instanceof rep Object;
        b = o instanceof readonly Object;

        /* cast */
        t.f = (peer Object) t;
        rep Object x = (rep Object) o;


        /* ------ arrays ------ */
        /* new */
        readonly readonly Object[] a = new peer peer Object[5];
        a = new rep readonly Object[5];

        /* instanceof */
        b = a instanceof rep peer Object[];
        b = a instanceof peer readonly Object[];

        /* cast */
        rep readonly Object[] c = (rep readonly Object[]) a;

        /* array store */
        peer readonly Object[] s = new peer peer test[6];
        s[1] = new peer test();


        /* ------ static functions ------ */
        bar();
    }

    public static void bar () {
        peer Object o = new peer Object();
    }
}
```

Listing C.19: Example Universe Java program used to show the effects of the sourcecode transformations (see listing C.20).

TODO: format JAD output & put checkArrayType back in...

```
// Decompiled by Jad v1.5.8e. Copyright 2001 Pavel Kouznetsov.
// Jad home page: http://www.geocities.com/kpdus/jad.html
// Decompiler options: packimports(3)
// Source File Name: test.java

import org.multijava.universes.rt.*;

class test {
    static Object sf;
    Object f;

    static {
        /* use the root context for static initialization */
        UniverseRuntime.handler.setContext(UniverseRuntime.handler.getRootSetMember());

        /* initialize static field */
        UniverseRuntime.handler.setConstructorData(UniverseRuntime.handler.getContext(), Class.
             forName("java.lang.Object"), 40);
        sf = new Object();
        UniverseRuntime.handler.setOwnerPeer(sf, UniverseRuntime.handler.getContext());
        /* static initializer code */
        UniverseRuntime.handler.setConstructorData(UniverseRuntime.handler.getContext(), Class.
             forName("java.lang.Object"), 40);
        Object o = new Object();
        UniverseRuntime.handler.setOwnerPeer(o, UniverseRuntime.handler.getContext());

        /* reset context stack */
        UniverseRuntime.handler.resetContext();
    }

    /* constructor */
    public test() {
        UniverseRuntime.handler.setOwner(this);
        Block$();
        UniverseRuntime.handler.setConstructorData(this, Class.forName("java.lang.Object"), 40);
        f = new Object();
        UniverseRuntime.handler.setOwnerPeer(f, this);
    }

    /* initialize non-static fields */
    private void Block$() {
        UniverseRuntime.handler.setConstructorData(this, Class.forName("java.lang.Object"), 40);
        f = new Object();
        UniverseRuntime.handler.setOwnerPeer(f, this);
    }

    public static void main(String args[]) {
        UniverseRuntime.handler.setConstructorData(UniverseRuntime.handler.getContext(), Class.
             forName("test"), 40);
        Object $tmp = new test();
        UniverseRuntime.handler.setOwnerPeer($tmp, UniverseRuntime.handler.getContext());
        ((test) $tmp).go();
    }

    public void go() {
        /* ----- objects ----- */
        /* new */
        UniverseRuntime.handler.setConstructorData(this, Class.forName("java.lang.Object"), 40);
        Object o = new Object();
        UniverseRuntime.handler.setOwnerPeer(o, this);

        UniverseRuntime.handler.setConstructorData(this, Class.forName("java.lang.Object"), 42);
        o = new Object();
```

```
UniverseRuntime.handler.setOwnerRep(o, this);

/* new universe aware object */
UniverseRuntime.handler.setConstructorData(this, Class.forName("test"), 40);
test t = new test();
UniverseRuntime.handler.setOwnerPeer(t, this);

/* instanceof */
Object $tmp = o;
boolean b = ($tmp instanceof Object) &&
    UniverseRuntime.handler.isPeer(this, $tmp);

$tmp = o;
b = ($tmp instanceof Object) &&
    UniverseRuntime.handler.isOwner(this, $tmp);

b = $tmp instanceof Object;

/* cast */
t.f = t;

$tmp = o;
if ( !UniverseRuntime.handler.isOwner(this, $tmp) )
    UniverseRuntime.policy.illegalCast();
Object x = $tmp;

/* ----- arrays ----- */
/* new */
Object a[] = new Object[5];
UniverseRuntime.handler.setArrayOwnerPeer(a, this, 40);

a = new Object[5];
UniverseRuntime.handler.setArrayOwnerRep(a, this, 41);

/* instanceof */
Object $atmp[] = a;
b = ($atmp instanceof Object[]) &&
    UniverseRuntime.handler.isOwner(this, $atmp) &&
    UniverseRuntime.handler.checkArrayType($atmp, 40);

$atmp = a;
b = ($atmp instanceof Object[]) &&
    UniverseRuntime.handler.isPeer(this, $atmp);

/* cast */
$atmp = a;
if ( !UniverseRuntime.handler.isOwner(this, $atmp) )
    UniverseRuntime.policy.illegalCast();
Object c[] = $atmp;

/* array store */
test s[] = new test [6];
UniverseRuntime.handler.setArrayOwnerPeer(s, this, 40);
UniverseRuntime.handler.setConstructorData(this, Class.forName("test"), 40);
s[1] = new test();
UniverseRuntime.handler.setOwnerPeer(s[1], this);
if ( !UniverseRuntime.handler.checkArrayType(s, 41) &&
        !UniverseRuntime.handler.isPeer(s, s[1]) )
    UniverseRuntime.policy.illegalArrayStore();

/* ----- static functions ----- */
UniverseRuntime.handler.setContext(this);
bar();
UniverseRuntime.handler.resetContext();
```

```
    }

    public static void bar() {
        UniverseRuntime.handler.setConstructorData(UniverseRuntime.handler.getContext(), Class.
            forName("java.lang.Object"), 40);
        Object o = new Object();
        UniverseRuntime.handler.setOwnerPeer(o, UniverseRuntime.handler.getContext());
    }
}
```

Listing C.20: Decompiled class file showing the effect of the sourcecode transformations to the program in listing C.19.