Runtime Support for Generics and Transfer in Universe Types

Mathias Ottiger

mathias.ottiger@gmx.net

Master Project Report

Software Component Technology Group
Department of Computer Science
ETH Zurich

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

February 2007 – August 2007

Supervised by:

Arsenii Rudich
Dipl.-Ing. Werner M. Dietl
Prof. Dr. Peter Müller

Software Component Technology Group

inf | Informatik
Computer Science

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract

The Universe type system allows a programmer to control aliasing and dependencies in object-oriented programs by applying an ownership relation to structure the object store. In previous projects the Universe type system was extended by static checks for Uniqueness and ownership transfer [[5], [6]] and for generic types in the Universe Type System [[1], [2]]. The main part of this master project is to design and implement the runtime support for Uniqueness (and ownership transfer) and for generic types. From Java 5 generic types are introduced. Java does not store the runtime type of the type arguments. But the Universe type system needs this information for its runtime model. Based on GUT [[2], [3]] which defines the runtime model for generic types in the Universe type system the current runtime implementation in the multijava [11] compiler is now extended by an implementation for this runtime model. The runtime support for Uniqueness is an extension of the basic Universe runtime implementation to fulfill the requirements of Uniqueness and ownership transfer.

Contents

1 Introduction

This chapter describes the concept of the Universe type system. It also defines the Universe type system with ownership transfer. Fur the Universe type system with ownership transfer the shorthand Uniqueness is used. This chapter also describes the concept of generic types in the Universe type system.

1.1 Motivation

The Universe type system allows a programmer to control aliasing and dependencies in object-oriented programs by applying an ownership relation to structure the object store. The Universe type system is built on multijava [**10**], an open source compiler which is a rebuilt of the Java 1.4 compiler.

1.2 Introduction to Universe Type System

In the Universe type system the creation of and access to objects are qualified by the modifiers **rep**, **peer** and **readonly**. Every object has an owner. The owner is an object too.
Objects created with the **rep** modifier are in the representation of the creator of the objects. The creator is the owner of these objects.

```
rep Object x = new rep Object();
rep Object y = new rep Object();
rep Object z = new rep Object();
```

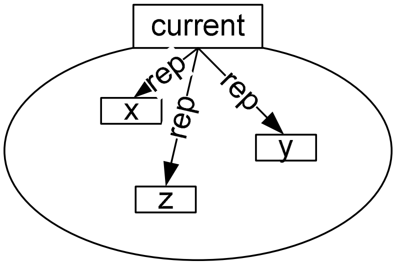


Figure 1-1: rep Objects of current – x,y and z are in the representation of current. The owner of x,y and z is current.

On the left hand side in the table above the code for creating rep objects is shown. For understanding reasons we always assume that the code is executed at runtime somewhere on an object called current. For the code above it means that current creates the objects x,y and z. As The owner of these created objects is current and therefore the objects are in the representation of current as the runtime model in Figure 1-1 shows.

Objects created with the **peer** modifier are in the same representation as the creator of the objects as illustrated in Figure 1-2. The owner of these objects is the owner of the creator of the objects.

```
peer Object x = new peer Object();
peer Object y = new peer Object();
```

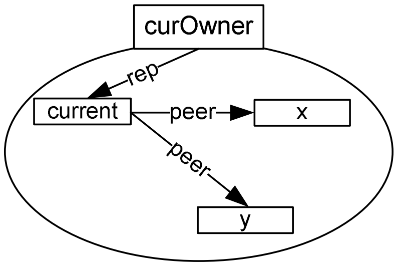


Figure 1-2: peer Objects of current – x and y are in the same representation as current. The owner of x and y is curOwner.

The code on the left hand side in the table above shows the creation of two peer object. Again the objects are created in the context of current. When creating peer objects the owner x and y is the same current has, which is curOwner as the runtime objects in Figure 1-2 show.

readonly Objects cannot be created. The **readonly** modifier is only used for references.

```
rep Object x = new rep Object();
readonly Object ro = x;
```

As this code shows a **rep** Object is created and assigned to x. In the second line a **readonly** reference ro is created which points to x. Over ro it is not possible to perform any writes onto ro's fields or to call any methods which perform write operations. Methods which do not have any write operations can be declared with the modifier **pure** which denotes that the method has no write operations.

1.3 Introduction to Uniqueness and Ownership Transfer

Uniqueness and Ownership Transfer is built on the current Universe type system. In this project report "Uniqueness" is used as an abbreviation for "Uniqueness and Ownership Transfer". Uniqueness allows the programmer to transfer objects from one representation to another. So far this was not possible in the Universe type system. To denote that objects can be transferred a new modifier called **uniq** is introduced. The **uniq** modifier is **rep** modifier whos owner can be changed.

```
class C {
    uniq Object x;

    void create(){
        x = new rep Object();
    }
}
```

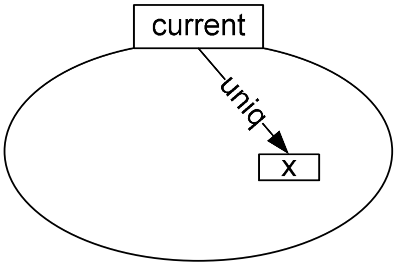


Figure 1-3: a uniq reference to x

In the table above the code for creating a **uniq** reference is shown on the left hand side. The runtime objects in Figure 1-3 show that x is in the representation of current and that current has a **uniq** reference to x.

Uniqueness does not only allow transferring single objects. Uniqueness is much more powerful and allows transferring sets of objects at one time. The representation of an object can now be divided into clusters. A cluster is a set of objects and is a part of a representation. All objects of one cluster can be transferred at one time.

```
class C {
    uniq Object x;
    rep[x] Object y;
    uniq Object z;
}
```

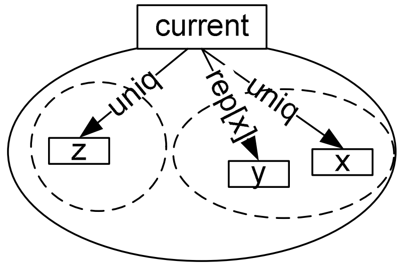


Figure 1-4: Uniqueness and clusters

The **uniq** modifier defines a cluster. So a cluster for x and for z is defined. The modifier rep[x] defines that y points into the same cluster as the **uniq** reference x does.

In Universe Types two objects are **peer** if they have the same owner. In Uniqueness two objects are peer if they are in the same cluster.

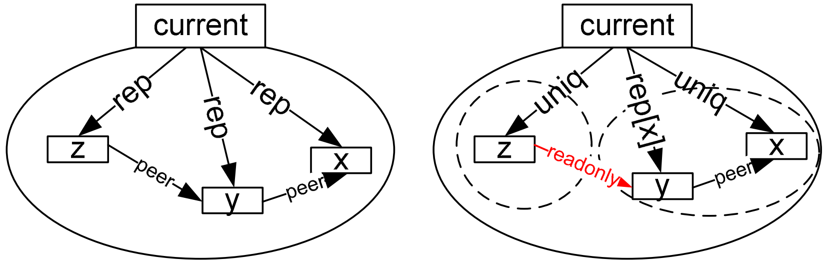


Figure 1-5: The definition of peer types in Universe Types (left) and in Uniqueness (right)

Figure 1-5 shows the difference of peer objects in the current Universe type system (left) and Uniqueness (right). As for the current Universe type system two objects are peer if both have the same owner the Uniqueness has more granularity because it expects that peer objects have to be in the same cluster too. Objects having the same owner and being in different clusters can have **readonly** references to each other.

Clusters can be merged and transferred. In the following the possibilities for transferring and merging cluster is described.

1.3.1 Merging Clusters

As already described objects being in different clusters are not **peer**. But if the programmer creates a **peer** reference from one object in a cluster to an object in another cluster the clusters of both objects are merged.

```
         class T {
             peer T next;
         }

         class A {
             uniq T x;
             rep[x] T y;
             uniq T z;

             public A() {
                 x = new rep T();
                 y = new rep T();
                 z = new rep T();
             }
             void merge(){
pc-->            y.next = z;
                 z = new rep T();
             }
         }
```

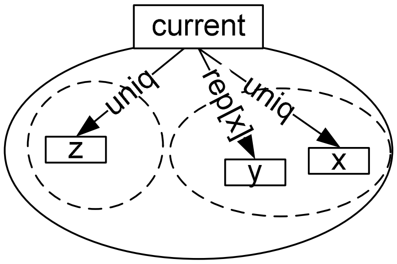


Figure 1-6: runtime objects before the execution of the line at pc

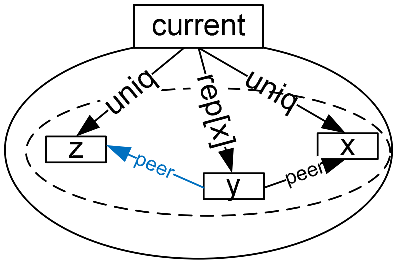


Figure 1-7: runtime objects during the execution of the line at pc

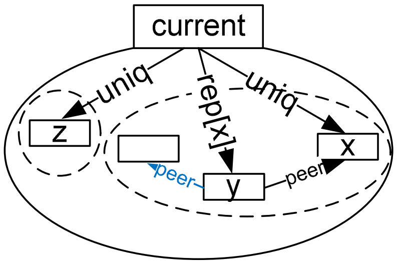


Figure 1-8: runtime objects after the execution of the line at pc

The table above show how clusters can be merged. On the left hand side pc shows the line of code which is executed at runtime. Figure 1-6, Figure 1-7, Figure 1-8 show the runtime objects with its clusters before, during and after the assignment y.next = z.

1.3.2 Transferring Cluster

In Uniqueness clusters can be transferred from **rep** to **peer** context as the following example shows.

```
         class Transferrer {
             uniq Object x;
             public Transferrer(){
                 x = new rep Object();
             }
             public void transfer(){
pc-->            peer Object y = x;
npc-->           x = null;
             }
         }
```

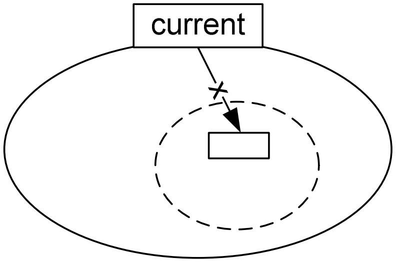


Figure 1-9: before execution of the code at line at pc

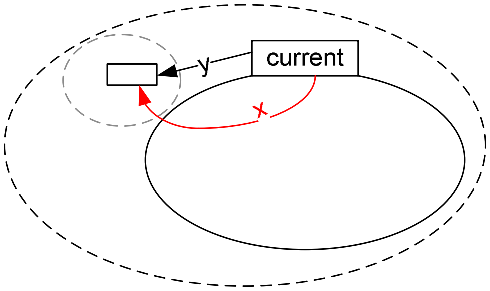


Figure 1-10: after execution of the code at line at pc, the reference x is unusable

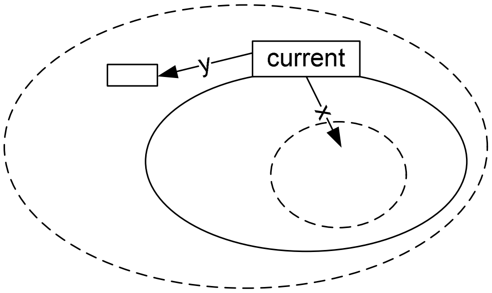


Figure 1-11: after execution of the code at line npc. x is now assigned to null and therefore x points into a new cluster.

This example shows the transfer of the **uniq** object x with its cluster to the **peer** context. Figure 1-9, Figure 1-10 show the runtime objects before and after the execution of the code at line pc. After this assignment x is set unusable (see dashed arrow) because the object where x points to is not in the **rep** cluster any more. Figure 1-11 shows the runtime objects after the execution of the code at line npc. Because x is assigned to null it does not point into the transferred cluster but points into a new cluster and therefore is not unusuable any more.

1.3.3 Capturing and Leaking

Another possibility of transferring objects from one cluster to another can be done by method calls where the programmer can define that the passed argument has to be transferrable (the cluster of the argument is transferrable). To define transferrable arguments (transferrable clusters) the modifier **free** is used.

```
     class T {
       uniq Object ref;
       void capture(free Object x){
         ref = x;
     }
     class TCaller {
       uniq Object f;
       public TCaller {
         f = new rep Object();
       }
       void callT(){
         rep Object arg = f;
         f = null;
         peer T t = new peer T();
pc→      t.capture(arg);
       }
     }
```

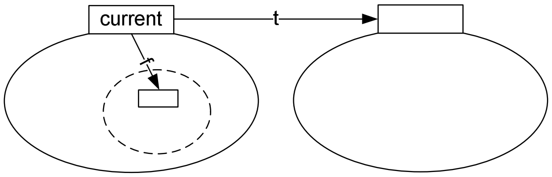


Figure 1-12: runtime objects before execution of the code at line pc

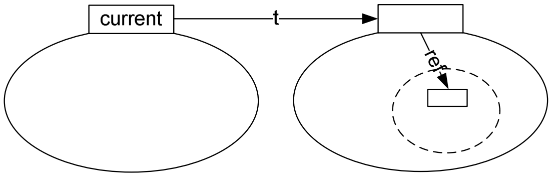


Figure 1-13: runtime objects after execution of the code at line pc

This example shows how a possible ownership transfer works by passing a free argument to a method. The runtime objects before and after the execution of the code at line pc are shown in Figure 1-12 and Figure 1-13.

The leaking mechanism works in a similar way as the capturing mechanism does.

```
        class S {
          uniq Object ref;
          S(){
            ref = new rep Object();
          }
          free Object leak(){
            Object ret = ref;
            ref = null;
            return ret;
          }
        }

        class SCaller {
          uniq Object f;
          void callS(){
            peer S s = new peer S();
pc-->       f = s.leak();
          }
        }
```

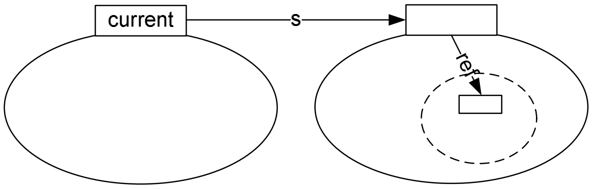


Figure 1-14: runtime objects before execution of code at line pc

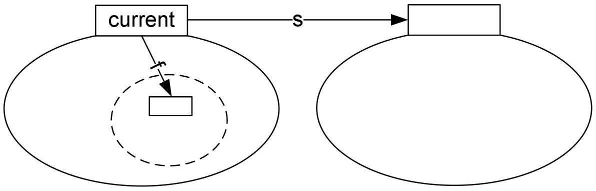


Figure 1-15: runtime objects after execution of code at line pc

This example shows how a possible ownership transfer works by getting a free return object from a method call. Figure 1-14 and Figure 1-15 show the runtime objects before and after the execution of the code at line pc.

1.4 Generic Universe Types

As the Universe type system is built on the multijava compiler which implements the Java 1.4 API generic types are not used yet. With Java 5 generic types [**8**] are introduced and the multijava compiler is extended by supporting generic types too. During this project the implementation for static type checks for generic types in the Universe type system [**4**] was finished. As the Universe type system needed additional runtime checks which are not covered by the standard java 1.4 API (therefore not covered by multijava either) generic types need runtime checks according to Generic Universe Types [[**2**], [**3**]]. As the runtime model defined by Peter Müller and Werner Dietl [[**2**], [**3**]] the Universe type system needs to keep track about the type arguments of generic created objects.

```
class Node<K,V>{
    Node/*<K,V>*/ next;
    public void test(){
        next = new peer Node<K,V>();
        System.out.println(next instanceof Node<peer Node<peer
                String,peer String>, peer Integer>);
    }
}

rep Node<rep Node<rep Integer,rep Integer>,rep String> x =
    new rep Node<rep Node<rep Integer,rep Integer>,rep String>();
x.test();
```

The code above shows a possible instanceof expression. With this project, the Universe type system provides runtime checks for the type arguments. The type arguments are now checked about its owners and class identifiers. A

```
x instanceof Node<rep X1, peer X2>
```

checks now if the first and second type arguments of x have owner equals this and owner of this. The modifiers of the type arguments are in the same view point as the main modifier is. The main modifier is the only modifier which is not in a type argument. Further the class identifiers of the first and second type argument of x are compared with X1 and X2. All the type rules about generic Universe Types can be written in the literature [[**2**], [**3**]].

For better understanding of the viewpoint of the type arguments, a second example is shown.

If the local variable k is defined as:

```
rep Node<peer Integer, rep String> k;
```

the modifiers with its class identifiers have the following meanings:

rep Node, where the rep is the main modifier, defines that the owner of the Node is this. peer Integer which is the first type argument defines that the owner of the type argument is the owner of this. rep String defines that the owner of the type argument String is this. Note that the owners of the type arguments are not relatively to the main modifier they are always absolute (in the viewpoint of this). The owner of rep String is not Node!

1.5 Overview

In the next chapter the runtime support for Uniqueness and generic Universe types are explained. Especially it is shown where runtime checks have to be done and where the runtime model has to be updated (for Uniqueness). In chapter two the concepts of the runtime support for the Universe type system is discussed. In chapter three the detailed description of the runtime model is discussed and in chapter four the implementation of this runtime model is explained.

2 Concepts of the Runtime Support for the Universe Type System

This chapter describes the main concepts for the runtime checks in Universe types as well as for Uniqueness. Further the concept of generic types in the Universe type system is explained. Especially class and method type variables and their use at runtime are discussed.

2.1 Current Universe Type System

For the runtime support for the Universe type system [**7**] every object has an owner. Having this information is enough for providing the runtime support. The owner of every object must be stored. The owner of an object is checked by instanceof expressions or by casts.

```
class T {
    void m(readonly Object x){
        if (x instanceof rep Node){
           rep Node first = (rep Node)x;
}
```

The code in the table above shows a possible instanceof expression. In addition to the standard checks of Java it has to be checked if x was created by the object where method m is called. This check is performed by comparing the owner of x with this. For the cast (rep Node)x in the next line x is checked again if its owner is this.

```
class T {
    void m(readonly Object x){
        if (x instanceof peer T){
           peer T next = (rep T next)x;
}
```

The code above shows the instanceof and cast expression if the peer modifier is used. Here x is checked whether its owner is same as the owner of this.

2.2 Uniqueness

In Uniqueness runtime checks are also needed for instanceof and cast expressions. But with the possibility of transferring clusters the current runtime checks have to be extended. For the runtime support for Uniqueness not the owner of every object must be stored but the cluster it belongs to. In the following subchapters it is shown how and when the transfer and merging of clusters take place.

2.2.1 Assignments

Let

$$x = y$$

be an assignment where x and y are local variables. As described in 1.3.2 (Transferring Cluster) assignments perform transferring cluster if the modifier of x is peer and the modifier of y is rep.

Assignments which fulfill this condition (where a rep is assigned to a peer) all objects in the cluster where y is, are transferred into the cluster where x is.

2.2.2 Method Calls on this

When we pass a rep Object x as an argument to a method called on this which takes a peer Object as parameter then all the objects in the cluster where x points into are transferred into the cluster where the object of the called method points into.

```
class Test {
    void m(peer Object x){}
    void doTransfer(){
        rep Object arg = new rep Object();
        m(arg);
    }
}
```

This example shows the method call m in the method doTransfer. The rep Object arg is passed as an argument to the method m which takes a peer Object parameter. The system performs the transfer of all objects where arg points into to the cluster where **this** is.

2.2.3 Method Calls Objects on Other Objects

The behavior of calling methods on rep or peer objects are different than calling methods on this.

2.2.3.1 Calling Method on peer Object s

Calling methods with peer parameters on peer objects perform an ownership transfer if the passed argument is of type rep. The passed rep object performs the transfer of all objects of the cluster where the passed argument points into to the cluster where the peer object we call the method points into.

```
class Caller {
    void doSth(peer Object p){}
}

Caller c = new peer Caller();
rep Object x = new rep Object();
c.doSth(x);
```

This example shows a method call on a peer object. A peer Caller c is created. The peer object is not equal this. Now a rep Object x is created and passed as argument to the call c.doSth. All objects where x points into are transferred into the cluster where c points into.

If the peer object where the method is called is equal this then the same behavior takes place as the method is called on this.

```
class Caller {
    peer Caller next;
    void doSth(peer Object p){}
    void doTransfer(){
        next = this;
        rep Object x = new rep Object()
        next.doSth(x);
}
```

This example shows the behavior of the method call doSth on peer Caller next where next==this. Because next is equal this the method call doSth performs the transfer of all objects where x points into to the cluster where this points into.

2.2.3.2 Calling Method on rep Objects with rep Parameter

If a method on a rep object is called which takes a parameter with a peer modifier and the argument's modifier is of type rep then the argument has to be in the same cluster as the object on which the method call is performed. Therefore a merge operation is performed so that the argument is in the same cluster as the object on which the method is called.

2.2.4 Instanceof Expressions

The instanceof expression in Uniqueness is defined similarly as for the Universe Type. With the instanceof expression we can check if an object is in a certain cluster too.

```
class T {
    uniq T f;
    void m(readonly Object p){
        if (p instanceof rep[f] T){
        }
    }
}
```

In this example the object p is checked, if it is of type T and if the object referenced by p is in the same cluster as f points into.

```
class U {
    void m(readonly Object p){
        if (p instanceof peer U){
        }
    }
}
```

In this example the object p is checked, if it is of type U and if the object referenced by p is in the same cluster as the object where m is called (if it is in the same cluster as this).

2.2.5 Cast Expressions

Cast expression's behavior is similar to the instanceof expression except that the latter returns false if the expression is not true where the former raises a cast expression.

```
class U {
    uniq U f;
    void m(readonly Object p){
        rep T x = (rep[f] U) p;
    }
}
```

This example shows the (rep[f] U) cast expression. If p is not in the same cluster as f or if p is not of type U a cast expression is raised.

```
class U {
    void m(readonly Object p){
        peer U y = (peer U) p;
    }
}
```

This example shows the (peer U) cast expression. If p is not in the same cluster as **this** or if p is not of type U a cast expression is raised.

Because Uniqueness allows to transfer clusters from rep to peer context the cast expression allows it too. If a rep object is casted to a peer object the cluster of the casted object is transferred into the cluster where **this** is.

```
class V {
    peer V next;
    uniq Object f;
    void m(){
        rep Object  x = f;
        f = null;
        next = (peer V) x;
    }
}
```

This example shows how an object is transferred from rep to peer context in a cast expression. In method m the uniq reference if is assigned to the local variable x and then set to null. Then x is casted to peer V. Because x is of type rep and the destination of the cast is peer the cluster where x points into is transferred where this points into.

So instanceof expressions never perform transferring clusters and cast expression only perform transferring clusters if the tested object is of type rep and if the object is casted to peer.

2.2.6 Field Writes

Let

$$x.f = y$$

be a field write expression where x is an object having a field f and y be an object. This field write expression could be used in a code fragment as follows:

```
class T {
    peer T f;
}
peer T x = new peer T();
rep T y = new rep T();
x.f = y;
```

As the Uniqueness [[**5**], [**6**]] defines assignments can cause cluster transfers if the right hand side fulfills the condition:

- The modifier on the right hand of the assignment must always be rep.

Depending on the modifiers of x and f the clusters can be transferred.

If the modifier of x is peer and the modifier of f is peer, the objects being in the cluster where y is are moved into the cluster where x.f points into. Note if the modifier of x is this which is a special case of peer then we have a simple rep to peer assignment which was already defined in 2.2.1 Assignments.

If the modifier of x is rep and the modifier of f is peer, the objects being in the cluster where y is are moved into the cluster where x.f point into.

2.3 Generic Types

2.3.1 Generic Types in Java

To understand how generic types work in Universe Types a short overview of the generic types in Java is shown. Generic types are introduced in Java 5 [**8**]. Classes can be defined wit type variables as K and V in the class List in the code below. The programmer can instantiate the class List by passing type arguments for the type variables K and V. A possible creation of an object of type List<K,V> can be:

```
new List<Integer,String>()
```

with this statement K is replaced by the type Integer and V by String in the class List. So every type variable is replaced by its type argument.

```
class List<K, V>{
    Node<K,V> first;
    void add(K key, V value){
        first = new Node<K,V>();
        first.key = key;
        first.value = value;
}

class Node<K,V>{
    Object key;
    Object value;
}

List<Integer, String> l1 = new
List<Integer,String>();
l1.add(new Integer( 1), “PIN”);
```

This code example shows how generic classes can be defined with type parameters K and V. The class List has type parameters K and V. These type parameters are used to create a Node with K and V in the method add. l1 is of type List with the type arguments Integer and String so l1 is created as Node<Integer,String>. With the add method a Node<Integer,String> is created and assigned to l1.first.

2.3.2 Generic Types in the Universe Type System

As already mentioned the type arguments in the Universe type system have modifiers too. As defined in Generic Universe Types [[**2**], [**3**]] the owner and the class identifier of the type arguments are needed for defining the runtime type of the type arguments. Java 5 does not store the runtime type of the type argument’s type. That means that when a List<K,V> is instantiated as:

x = new List<Integer,String>

it cannot be tested at runtime if x was instantiated with the type arguments Integer and String. This information is not stored in the runtime model.

Therefore a model for the runtime types of Universe generics objects has to be implemented from scratch.
A generic class in the Universe type system looks as follows:

```
class GenericClass<A,B,C,D>{}
```

A possible creation of this class can be:

```
new peer GenericClass<rep Integer, peer String,
                      peer Integer, rep String>()
```

As defined in Generic Universe Types the owner of every type argument has to be stored and the class identifiers of the type arguments are stored too. This information is enough for representing the runtime types of generic objects. That means the modifiers for the generic types are replaced by its owners at runtime. The modifiers themselves therefore are not stored at runtime. The modifiers are used for static checks.

But how to create and to get these type arguments information need some more detailed descriptions. The following steps show what has to be done to update the runtime model.

```
class Item<X>{}

rep Item<rep Integer> i = new rep Item<rep Integer>();
```

As the code above shows an object of type rep Item<rep Integer> is created. We assume that the code in the last line is executed in an object and its reference is this. Because the modifier of the type variable X is rep, the owner “this” has to be stored for the type variable X. The class id Integer of the type variable of X is stored too.

The code in the table below shows the use of class type variables.

```
class Item<X>{
    Item<X> next;
    public Item(){
        next = new Item<X>();
    }
}

rep Item<rep Integer> i = new rep Item<rep Integer>();
```

As before the type arguments information for the object of type rep Item<rep Integer> are stored. By creating this object, the constructor Item creates already new Item<X> which is assigned to next. The new Item<X> uses the type variable X. So the type of X is not known statically. That means that at runtime the runtime type of X must be found out and the type variable X must be replaced by the found runtime type. Therefore creating generic objects having type variables need a look up to find out the runtime type of the type variables.

For method type variables a look up is needed too. The code below shows a possible method call where the method uses method type variables:

```
class Node<K,V>{}

class NonGeneric {

    <K,V> void createNode(K key, V value){
        Node<K,V> n = new Node<K,V>();
    }

    void test(){
        createNode(new Integer(1), "aString");
    }
}
```

The method createNode reuses the method type variables K and V for creating a Node object with the type arguments K and V. The runtime types of K and V have to be found out at runtime.

2.3.3 Instanceof Expression

An instanceof expression can be as defined in the method check in the code below.

```
class Node<K,V>{}

class Checker {
    boolean check(peer Object x){
        if (x instanceof peer Node<rep Integer, peer String>){
            return true;
        }
        return false;
    }

    void startTest(){
        peer Node<rep Integer, peer String> n =
            new peer Node<rep Integer, peer String>();
        check(n);
    }
}
```

The instanceof expression in the method check checks if x is a generic type and if the owners of the type arguments (rep and peer from x **instanceof** peer Node<rep Integer, peer String>) are the same as those of x. The class ids (Integer and String) of the type arguments are compared too. One could think that it is not necessary to store the owners of the type arguments instead it is enough to only store the modifiers itself. The reader can look up an example in the appendix which shows that storing the owners is needed.

2.3.4 Cast Expression

The same concept as already illustrated in 2.3.3 Instanceof Expression is reused.

2.3.5 Type Arguments with the readonly Modifier

Non generic types can be created with the modifier peer or rep but not with the modifier readonly.

As defined in Generic Universe Types [[**2**],[**3**]] generic types can be created with the readonly modifier in the type arguments (but the readonly modifier is not allowed as main modifier). Under certain conditions as defined in 3.3 Subclassing and Subtyping in Generic Universe Types [**3**] the modifier readonly **can** be a super type of the modifier rep and peer. Chapter “4.1 Heap Model” of Generic Universe Types describes that if the modifier readonly is used for type arguments the owner of the readonly modifier is replaced by a special readonly object. In this project the object is called READ_ONLY_OBJECT. As the literature describes the READ_ONLY_OBJECT compared with other owner objects has special rules. If owners are compared as in casts or instanceof expression the result is the following.

Let us define the instanceof expression as follows:

$$X\ instanceof\ Y$$

Further let x be a type argument of X and y the corresponding type argument of Y. The equality of x and y

$$equalTypeArgument(x, y)$$

is defined as:

$$equalTypeArgument(x, x)\ =\ true$$

$$equalTypeArgument(x, READ\_ONLY\_OBJECT)\ =\ true$$

$$equalTypeArgument(x, y)\ =\ false\ otherwise$$

3 Runtime Models

3.1 Universe type System

For the runtime checks for the Universe type system the system must keep track of the owner of every object. The first approach was to extend the class java.lang.Object with a field owner. This extension raised problems with the Java virtual machine. Because of these problems, a global hash table was created which keeps track of every object's owner. To find out, who the owner of an object x is, a hash table lookup is needed to find out, who the owner of x is.

```
class List {
   rep Object n;
   public List(){
      n = new rep Node();
   }
}
```

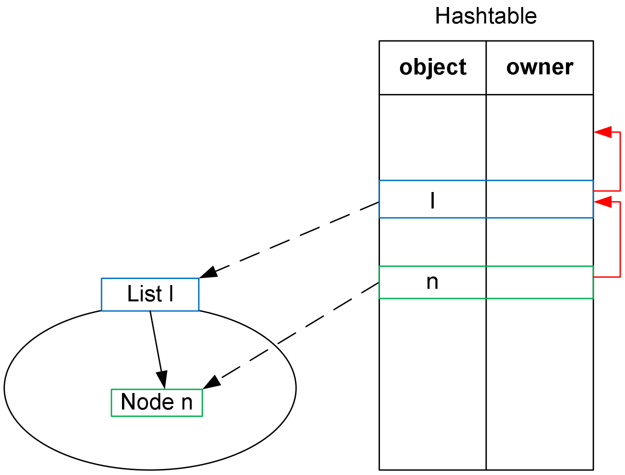


Figure 3-1: Universe type system runtime model, the Hashtable is a global defined table which keeps track of the relation of objects and its owners

When a List l is created then the entry [l, l's owner] is added to the hash table. When rep Node n is created then the entry [n,l] is added to the hash table. The dashed arrows define weak references (used for garbage collection purposes). The full arrows define references. This model is already implemented by Daniel Schregenberger [**7**].

As already described objects in Uniqueness belong to clusters and do not have an owner any more. So the global hash table was changed in order to store the information to which cluster an object belongs to. Before going deeper into detail of the global hash table it must be defined how the cluster can be designed.

3.2 Clusters

A cluster is a set of objects. Every object is assigned to a cluster. Clusters can be merged with other clusters. Or explained in other words all objects being in one cluster can be transferred into another cluster. In order to minimize merging cluster overhead the "Set Union Find" [**9**] mechanism is applied.

3.2.1 Set Union Find

Moving elements from one set to another cause a lot of operations. The more elements a set has the more time is needed to move all elements to another set.
Let us define Element and Set as follows:

```
class Element {
    Set set;
    public Element(Set aSet){
        set = aSet;
    }
}
class Set{}
```

Now two Sets set1 and set2 are created and the elements x1..x8 are assigned to the sets set1 and set2 as follows:

```
Set set1 = new Set();
Set set2 = new Set();
x1 = new Element(set1);
x2 = new Element(set1);
x3 = new Element(set1);
x4 = new Element(set1);
x5 = new Element(set2);
x6 = new Element(set2);
x7 = new Element(set2);
x8 = new Element(set2);
```

Now x1, x2, x3 and x4 are belong to set1 and x5, x6, x7, x8 belong to set2. Now we want to unify set1 and set2. That implies to perform the following steps:

```
x1.set = set2;
x2.set = set2;
x3.set = set2;
x4.set = set2;
```

That means the more elements belong to set1 the more operations are needed to unify set1 and set2.
The data structure for the set union find mechanism is used in this project as follows.

```
public class Set {
    Set next;
    public Set getLast(){
        if (next==null){
            return this;
        } else {
            Set cursor = next;
            while (cursor.next!=null){
                cursor = cursor.next;
            }
            return cursor;
        }
    }

    public void merge(Set other){
        Set ofThis;
        Set ofOther;
        ofThis = getLast();
        ofOther = other.getLast();
        if (ofThis != ofOther){
            ofThis.next = ofOther;
        }
    }
}

public class Element {
    Set set;
    public Element(){
        set = new Set();
    }
    public void unify(Element other){
        set.merge(other.set);
    }
}
```

With this data structure x1..x8 can be created as follows.

```
x1 = new Element();
x2 = new Element();
x3 = new Element();
x4 = new Element();
x2.unify(x1);
x4.unify(x3);
x3.unify(x1);
x5 = new Element();
x6 = new Element();
x7 = new Element();
x8 = new Element();
x6.unify(x5);
x8.unify(x7);
x7.unify(x5);
```

When creating an element a corresponding Set object is created to. If two elements xi and xj are in the same Set then xi.set.getLast() returns the same Set as xj.set.getLast() does. With the method call x2.unify(x1) we put x1 and x2 into the same set. After these operations described in the table above x1, x2, x3 and x4 are in the same set, as well as x5, x6, x7 and x8 do. If we want to unify the set where x1 is with the one where x5 is we do not have to update the references to the elements' set of x1, x2, x3 and x4 anymore. It is sufficient to perform the operation x7.unify(x5) which only has to update one reference in the data structure of the class Set. Of course the getLast method can perform many operations but with some optimizations the execution time of getLast can be speed up. This concept is used in this project for modeling the relation between objects and its clusters in Uniqueness.

3.2.2 Owner Objects Represent Cluster

For the runtime model for uniqueness the set union find algorithm is used. A cluster is defined as a set. The set is an object of type Owner. When creating any object a Owner object is created too. If two objects are in the same cluster their Owner object is connected. Connected Owner objects represent a cluster.

```
class C {
    uniq Object f;
    rep[f] Object g;
    uniq Object h;

    void init(){
        f = new Object();
        g = new rep Object();
        h = new rep Object();
    }
}
```

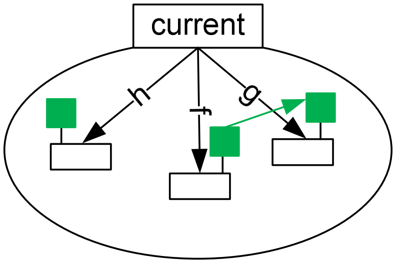


Figure 3-2: Runtime objects with its Owner objects

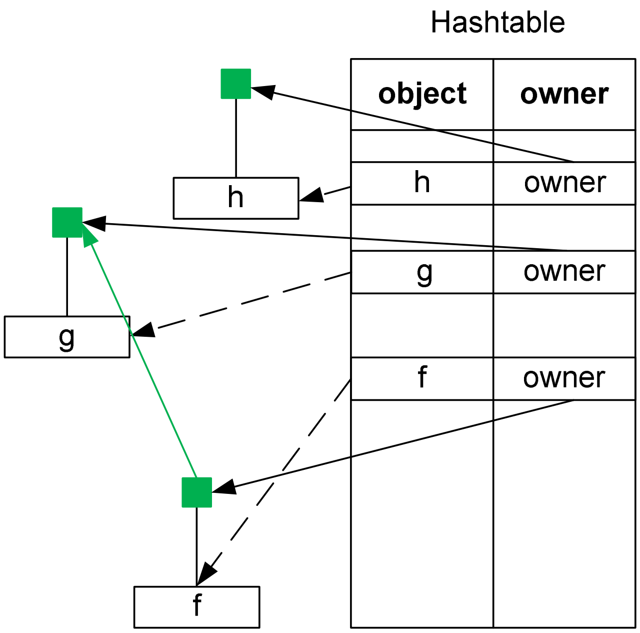
The init() method illustrates that for each created object referenced by f, g and h an additional Owner object (see the green rectangles in Figure 3-2 and Figure 3-3) is added. f and g as class C defines are in the same cluster so their Owner object is connected. h is in a separate cluster so there the Owner object of h is not connected with the Owner of f or g.

3.3 Uniqueness

The global hash table from the Universe type system is reused (see chapter 3.1). But the hash table entry was changed. The entry now stores the information of the object and its Owner object.

```
class C {
    uniq Object f;
    rep[f] Object g;
    uniq Object h;

    void init(){
        f = new Object();
        g = new rep Object();
        h = new rep Object();
    }
}
```



Hashtable

object	owner
h	owner
g	owner
f	owner

Figure 3-3: Uniqueness Hashtable

The hash table keeps the reference to the objects f, g and h and their Owner objects. Object f and g are in the same cluster so the Owner of f is connected with the Owner of g. The dashed arrows define weak references (used for garbage collection purposes). The full arrows define references.

3.4 Generics

3.4.1 Main Idea

The main idea for the runtime model was to change the current runtime model as it was implemented by Daniel Schregenberger [**7**] as few as possible. For Uniqueness no static type checks for generic types are implemented yet. There is no specification either. So the runtime model for generic types has to be designed primarily for the current Universe type system. But this model has to provide an interface for supporting the runtime support for generic types in Uniqueness too.

As already mentioned in 2.3 the owners and the class identifiers of the type arguments of every object must be stored at runtime. The runtime model must be designed so that the user can decide at runtime how the type arguments have to be checked. The user should be able to switch on and off checking type arguments' class identifiers.

3.4.2 Applying the Main Idea with the Current Runtime Implementation

The first idea was to reuse the basic hash table by extending the hash table entries by a field which refer on type arguments. This will have the effect that non generic objects have this field too and that this field is always null. We assume that most of the allocated objects are not generic and therefore most time this field is not used. Therefore we did not want to use this idea of extending the current hash table entry with one additional field.

Another solution which was taken into account was creating a subtype of hash table entries which store the type arguments information. But this concept was not used because of different reasons. Using a subtype of the hash table entries implies creating a subtype of the hash table. The interface for accessing and storing type arguments has many differences with the interface of the current hash table. Hence reusing the idea of the current hash table for a new hash table without any subtype relation has the advantage that less code changes in the current hash table implementation have to be performed. The only changes to the current hash table is now an additional implementation of the GenericsSupport (needed for the type arguments' owner, will be explained later in this report) interface. Now the current runtime implementation could be swapped out by another implementation which simply has to implement the GenericsSupport interface.

Because of these reasons and of minimizing the memory and time overhead a second hash table was created for keeping the information about type arguments. Non generic objects do not need the second hash table and have no time and memory losses.

```
class Item<K,V>{
    K key;
    V value;
}

class C {
    void doSth(){
        rep Item<rep Integer, peer String> x =
            new rep Item<rep Integer, peer String>();
    }
}
```

Imagine we want to store the type arguments information from the code above. After the execution of doSth the runtime model for generic types looks as follows:

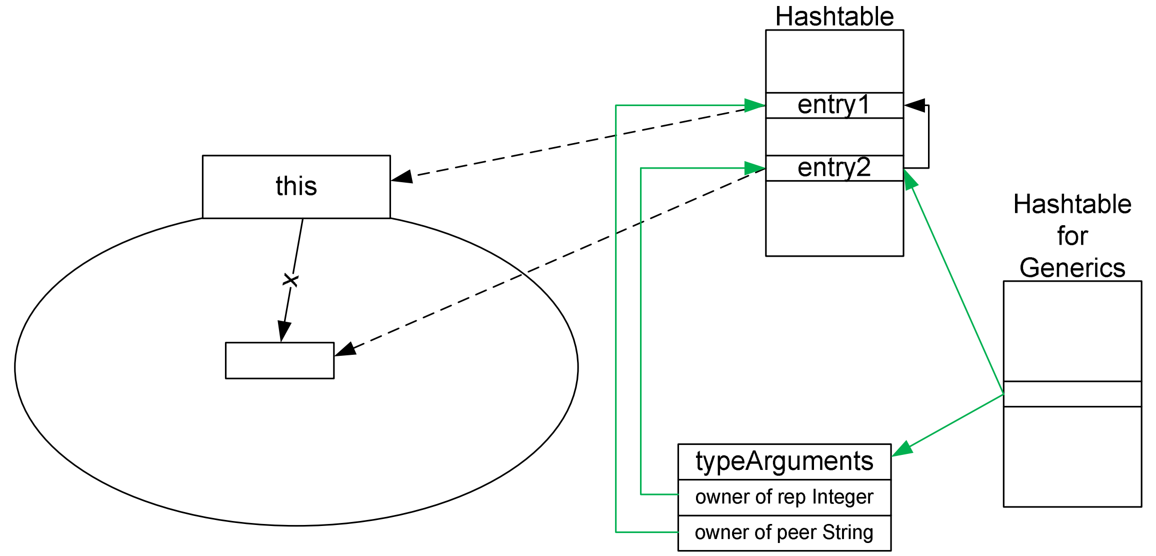


Figure 3-4: Generics Hashtable

Creating the rep x stores the information about x and its owner in the hash table. The entry for the type arguments is stored in the hash table for generics. This entry has no reference to this or to x. It only stores the reference to **entry2** (which hold the weak reference to x) and the reference to typeArguments (which holds the information about the type arguments). This is done in order to not prevent the garbage collector from collecting this or x. The owner of each type argument is stored as a reference to the entry of the first hash table which holds the weak reference (dashed arrow) to the owner. The first type argument of x which is “**rep Integer**” needs to store this as owner for the first type argument, so typeArguments stores the reference to the hash table entry **entry1**. The second type argument “**peer String**” stores the reference to the hash table entry which holds the weak reference to the owner of this (not shown in the graphic).

3.5 Garbage Collection

3.5.1 Cleaning Up Hashtable

As already described the Universe type System uses a hash table which stores the relation about object and its owner. If the hash table stores references to List l and Node n in a hash table entry, l and n could never be collected by the Java garbage collector unless the hash table entry is collected. This implies that we have to check hash table entries if they are not needed anymore. That means that we have to traverse all hash table entries to find out if they can be removed from the table. This will cost a lot of overhead and therefore is not used in our hash table.

Java provides weak references which are references too but their destination can be collected. To this weak reference we can assign a reference queue where weak references are put if their destination is collected. This concept is much more efficient because no table traversal is needed.

As already defined the hash table stores for each entry

- a weak reference to the object we want to save its owner
- a reference to another hash table entry which holds the weak reference to the owner object we want to store

If a Node n is created in the context of List l

- a new hash table entry is created where a weak reference to n and
- a reference to the hash table entry (which has the weak reference on l)

are stored.

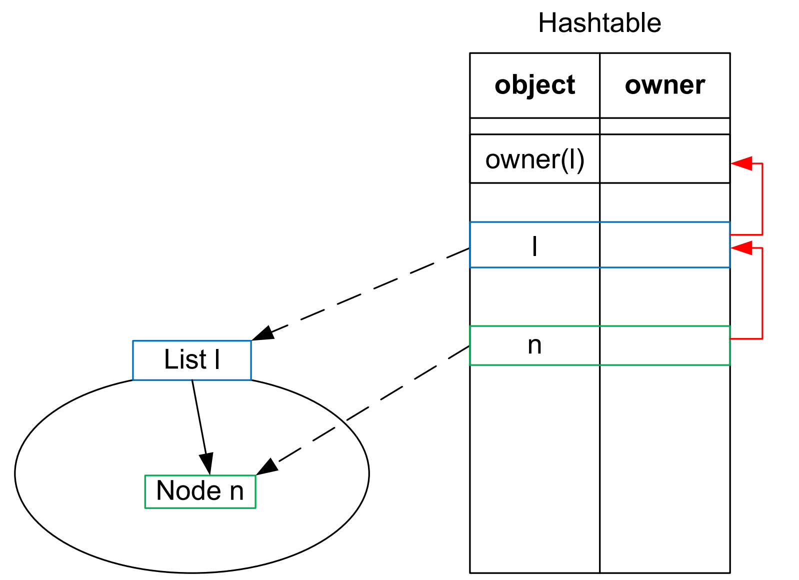


Figure 3-5: Current Hashtable

Let us define:

$$htEntry(x): the\ hash\ table\ entry\ which\ holds\ the\ weak\ reference\ to\ object\ x$$

With this concept l and n can be collected by the garbage collector. But if l is collected the $htEntry(l)$ cannot be removed because it may be needed by $htEntry(n)$ for instanceof or cast expressions. For this reason $htEntry(l)$ cannot be removed until all hash table entries which refer to $htEntry(l)$ are removed first.

A java.lang.ref.ReferenceQueue is used which contains references to weak references whose destination is already collected.
If l is collected by the carbage collector the weak reference with the destination to l is stored in the reference queue. The queue is regularly looked up (always on adding new hash table entries). If a weak reference is found in the reference queue it will be checked if the hash table entry can be removend from the table or not. The entry is still used as long as other hash table entries have references on it.
This is shown by an example where List l and Node n is reused from the figure above:

```
class Node{}
class List {
    void foo(){
        rep Node n = new Node();
    }
}
rep List l = new rep List();
l.foo();
l = null;
```

As this code shows, a List l is created. l creates a rep Node n with the method call l.foo(). Two hash table entries are created. In the last line of the code above l is set to null. Now l and n can be collected. Assume l and n are collected. Then they are added to the reference queue. The reference queue once is checked and finds the entry for l. The hash table is checked whether $htEntry(l)$ can be removed. It cannot be removed because a reference from $htEntry(n)$ still exists. The next entry is taken which is the one for n. $htEntry(n)$ can be removed from the hash table. Now the entry $htEntry(l)$ where $htEntry(n)$ points to is also checked because $htEntry(l)$ might be removed too if $htEntry(n)$ was the last entry which pointed to $htEntry(l)$. In this case $htEntry(l)$ can be removed now.

For keeping the number of hash table entries referring on a hash table entry every hash table entry contains a child count for storing the number of incoming references from other hash table entries.

This concept was already implemented in previous project for Universe type system. But this concept will be reused for Uniqueness and for Generics with some modifications.

3.5.2 Cleaning Up Hashtable in Uniqueness

In Uniqueness a hash table entry does not refer to another one. Because the hash table entry contains the weak reference to its object a reference to an Owner object (see green rectangle in the figure below).

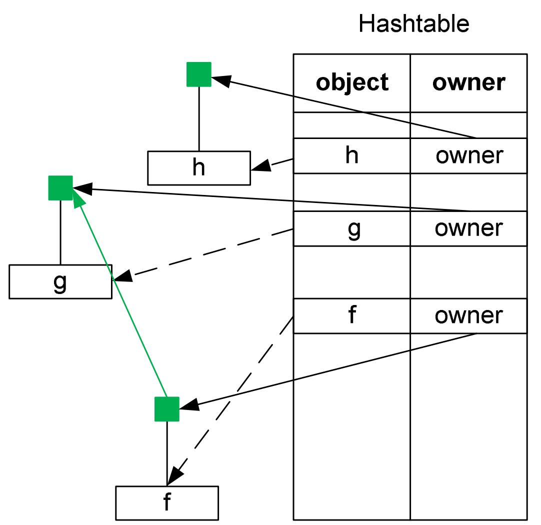


Figure 3-6: Uniqueness Hashtable

As the figure shows the objects f, g and h are stored in the Hashtable. f and g are in the same cluster because their Owner objects are connected. If g is collected $htEntry(g)$ can be collected too. The Owner object of g is not collected, but this has no influence on the Hashtable. This concept implies that when an object f is collected an entry to the reference queue is added for f and then $htEntry(f)$ can be removed without any further constraints. Because no more Owner object refer to the Owner object which was referenced by $htEntry(f)$ the Owner object can be collected by the garbage collector too.

3.5.3 Cleaning Up GenericsHashtable

For generic types a second hash table is used which keeps the information about the type arguments. The concept with the weak reference and the reference queue is reused.

```
class List<K>{
    rep Node<K> first;
    void init(){
        first = new rep Node<K> ();
    }
}
rep List<peer String> l = new rep List<peer String>();
l.init();
```

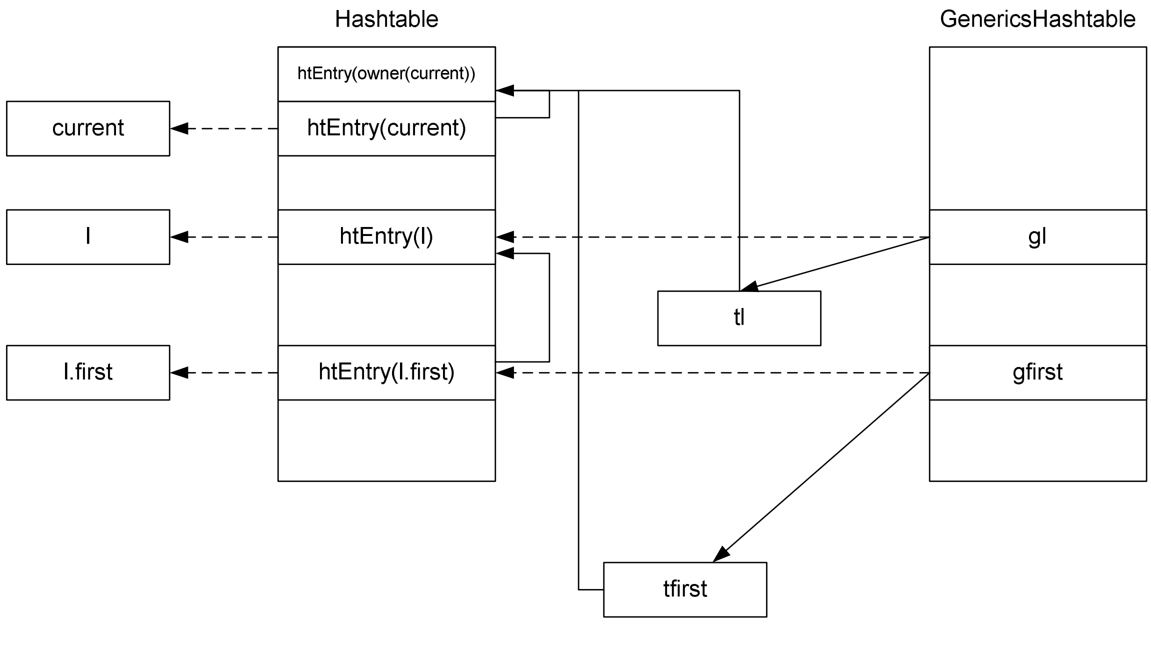


Figure 3-7: Generics Hashtable

The code in the figure above shows the references and weak references built in the Hashtable and GenericsHashtable. The object current represents the object where the last two lines of code are executed. When the rep List<peer String> l is created $htEntry(l)$ is created with the weak reference to l and a reference to $htEntry(current)$ which stores the owner of l. The type argument information for l is stored in the GenericsHashtable entry gl which has a weak reference to $htEntry(l)$ and a reference to a TypeArguments object tl. tl contains the information about the owners and class identifiers of all type arguments of l. The owners in the type arguments are identified by references to the Hashtable entries. In the code above the Node n is created with the same type argument as List l has. Therefore the owner of the type argument of first must be the same as the one of l. The entry for the arguments is stored in gfirst which has a reference to its TypeArguments object tfirst. The owner of the only type argument of first is the same as the one of l and for this reason tfirst and tl point to the same Hashtable entry where the weak reference to the owner of current is stored. The advantage with concept that owners of type arguments are represented as references to hash table entries is that the objects created by the programmer (as l and n) can be collected by the garbage collector. If the entries in the Hashtable are not needed any more, they can be removed from the Hashtable independently if there are any references from GenericsHashtable (or from its TypeArguments' references). If $htEntry(l.first)$ is removed from Hashtable htEntry(l.first) is still alive, but not registered in the Hashtable anymore. $htEntry(l.first)$ can be collected by the garbage collector. If this takes place, gfirst can be removed from GenericsHashtable too.

Another open issue is: how can owners of TypeArguments be compared if the entries in the Hashtable are already removed? Removing hash table entries from Hashtable do not kill the entry object as long as it cannot be collected. As the TypeArguments have normal references to the hash table entries of Hashtable they will not be collected until the TypeArguments are collected. And the TypeArguments are collected when the entry in the GenericsHashtable referring to the TypeArguments is removed from GenericsHashtable and collected.

4 Implementation

This chapter shows the implementation of the described runtime models from the previous chapter. All the implementation is done on the multijava [[**10**], [**11**]] open source compiler. Implementations are done in the **org.multijava.universes.rt** package and in its subpackages **org.multijava.universes.rt.impl** and **org.multijava.universes.rt.impl.generics**. Changes on the compiler itself is done in the package **org.multijava.mjc**. The testcases for the whole implementation is stored in the packages **org.multijava.mjc.testcase.universes.runtime.uniqueness** and **org.multijava.mjc.testcase.universes.runtime.generics**.

4.1 Uniqueness

4.1.1 Packages

The **org.multijava.universes.rt** package is extended with the following intefaces:

- **Owner**: Interface for the Owner objects
- **UniqImplementation**: reuse of the UrtImplementation, defines the operations for the runtime support for Uniqueness
- **UniquenessRuntime**: reuse of the UniverseRuntime, contains a static initializer which loads an implementation of the UniqImplementation. The default implementation is **UniqDefaultImplementation** (described later). The loaded implementation is accessible over org.multijava.universes.rt.UniqImplementation.handler which is used by the multijava compiler for maintaining the runtime support for Uniqueness. If the user wants to run any class with its own UniqImplementation the user can set the system property UniqImplementation to its own implementation. The user can run its program as follows:
  with the UniqDefaultImplementation:
  java x
  with its own UniqImplementation personalImpl:
  java –DUniqImplementation=personalImpl x

The **org.multijava.universes.rt.impl** package is extended with the following classes:

- **OwnerImpl**: the implementation of Owner, uses the Set Union Find algorithm for representing objects being in the same clusters
- **UniqDefaultImplementation**: the implementation of the UniqImplementation interface.
- **UniqHashtable**: reuse of UrtHashtable, stores now UniqHashtableEntry (explained later) instead of UrtHashtableEntry.
- **UniqHashtableEntry**: reuse of UrtHashtableEntry but stores now the OwnerImpl as owner of object.

4.1.2 Source Code Transformation

In order to provide the runtime support we have to add source code to the existing code we are compiling. In this chapter all imported transformations are explained which are needed to provide the runtime support for Uniqueness.
For overview reasons we use
handler=org.multijava.universes.rt.UniqImplementation.handler
as abbreviation.

4.1.2.1 Object Creation

During object creations the source code transformation works the same way as the Universe type system is used. The only difference is that in Uniqueness the Owner object is created too, but this is performed in the called method of *handler*.

```
class X {}                        class X {
                                      X(){
                                          handler.setOwner(this);
                                      }
                                  }
class T {                         class T {
    void doSth(){                     T(){
        rep X x = new rep X();            handler.setOwner(this);
    }                                 }
}                                     void doSth(){
                                          handler.SetOwnerRep(…);
                                          X x = new rep X();
                                      }
                                  }

peer T t = new T();               handler.setOwnerPeer(this);
t.doSth();                        T t = new T();
                                  t.doSth();
```

When creating an object we know the address of the object after execution of the object creation. Storing the owner of x after the object creation raises problems when x creates another objects in its constructor because then the object x is not registered in the hash table yet, but the object reference is already needed. Further details on this topic can be read in Runtime Checks for the Universe Type System [**7**]. So before creating any objects the *handler.setOwnerRep* or *handler.setOwnerPeer* (for creating peer objects) method has to be called. These methods store the information that the next created object is of type rep or peer and who the owner of the object is. So the first statement in every constructor is *setOwner* which gets the stored information and creates then the entry in the hash table. Of course in Uniqueness it is not important who the owner of an object is because for every created object a corresponding Owner object is created too. But the old information (that t is the owner of x) is still stored but not used in this project. This information is stored for further researches or verifications on Uniqueness.

4.1.2.2 Assignments

As already defined in 2.2.1 clusters have to be merged. The compiler checks whether it is needed or not. If not needed, the compiler does not perform any changes. If needed the compiler makes the following transformation:

```
class T {
  void doSth(){

    rep Object x = new rep Object();

    peer Object y = x;
  }
}
```

```
class T {
  void doSth(){
    handler.SetOwnerRep(this);
    Object x = new Object();
    handler.SetOwnerPeer(this);
    Object y = x;
    handler.mergeOwner(this,x);
  }
}
```

So the compiler merges the owner of the left hand side with the owner of the right hand side. So the owner of this and x have to be merged.

4.1.2.3 Clusters

When creating an object a corresponding Owner object is created too. If two objects are created which have to be in the same cluster their Owner objects have to be merged latest before leaving the method or calling another method.

```
class T {
    uniq Object f;
    rep[f] Object g;
    void createObjects(){
        f = new rep Object();
        g = new rep Object();
    }
}
```

As method T.createObjects shows, two Objects f and g are created. From 4.1.2.1 we know that the Owner objects are created. But f and g are not in the same cluster yet. For this purpose at the end of createObjects f and g have to be merged. For every method it has to be checked if there are declared fields which have to be merged. So the compiler checks this and generates the corresponding method calls as follows:

1. The compiler checks if the method is pure (no write operations, see XXX). If the method is pure, nothing has to be done. If method is not pure go on. Pure methods are defined with the pure modifier in the method signature as **void pure doNothing()**.
2. Get all class fields and check which fields have to be in the same cluster. For each found cluster, create an array holding all fields which refer to this cluster.
3. For each created array *clusterSet*, create a handler.mergeOwnerSet(*clusterSet*) method call. The mergeOwnerSet will merge all fields defined in clusterSet.

`class X {`
`    uniq Object a;`
`    rep[a] Object b;`
`    rep[a] Object c;`
`    uniq Object d;`
`    uniq Object e;`
`    rep[e] Object f;`
`    void doSth(){}`
`}`
	1.
	pure?
		2.
		get fields in same cluster
			clustersets:
			
			[a, b, c]
			[e, f]
				3.	*handler.mergeOwnerSet([a,b,c])*
					handler.mergeOwnerSet([e,f])

This example shows the steps for creating the *mergeOwnerSet* method calls.

Cell 1: The source code
Cell 2: Check if the method doSth is declared as pure. Not declared as pure, go on.
Cell 3: find out which fields have to point into the same cluster.
Cell 4: Found fields being in same cluster. Create array for each cluster.
Cell 5: Create the *mergeOwnerSet* method calls
Cell 6: the created method calls

It does not make sense to insert the created mergeOwnerSet method calls at the end of every method body because it is not guaranteed that the end of the method body is reached in every case. If there is a return statement before the end of the method body, the system will not execute the mergeOwnerSet operations. So the mergeOwnerSet calls could be added before every return statement. But if the method raises for example an unhandled exception the mergeOwnerSet calls are not executed either. The mergeOwnerSet calls always have to be executed regardless if there are any return statements before the end of the method body or exception can raise. To fulfill these requeirements the method body is put into the try block of a try-finally statement. The mergeOwnerSet calls ar put into the finally statement.

```
class X {
    uniq Object a;
    rep[a] Object b;
    rep[a] Object c;
    uniq Object d;
    uniq Object e;
    rep[e] Object f;
    void doSth(){
        a = new rep Object();
        b = new rep Object();
    }
}
```

```
class X {
    Object a;
    Object b;
    Object c;
    Object d;
    Object e;
    Object f;
    void doSth(){
        try {
            handler.SetOwnerRep(this);
            a = new Object();
            handler.SetOwnerRep(this);
            b = new Object();
        } finally {
            handler.mergeOwnerSet([a,b,c]);
            handler.mergeOwnerSet([e,f]);
        }
    }
}
```

This example shows the source code transformation for a method body in order to merge the class fields which have to refer into the same cluster. At the end of doSth a and b refer into the same cluster.

If class fields are null there is no Owner for these class fields and mergeOwnerSet will not try to merge these fields.

4.1.2.4 Method Calls

As already described the parameters of a method have to be compared with the passed arguments. If a method has peer parameter and we pass a rep, the cluster of the argument has to be merged with the owner of the object where to method is called. Passing a rep object to a peer object with peer parameter or passing a rep object to a rep object with peer parameter also perform cluster merging.

```
class T {                               class T {
    void doSth(peer Object x){}             void doSth(Object x){}
}                                       }

                                        handler.SetOwnerPeer(this);
peer T t = new T();                     T t = new T();
                                        handler.SetOwnerRep(this);
rep Object x = new rep Object();        Object x = new Object();
t.doSth(x);                             handler.mergeOwner(x,t);
                                        t.doSth(x);
```

This example shows one possible code transformation for method arguments which have to be merged. So the called method has a peer parameter and a rep argument is passed. The handler.mergeOwner method call is added.

As described in the previous chapter at the end of each method the mergeOwnerSet calls must be executet to be sure that objects which are declared as being in the same cluster are in the same cluster. But the mergeOwnerSet must be called before the method call t.doSth is executed. Why this operation is needed can be shown with the following example:

```
class T {                             class T {
    uniq Object f;                        Object f;
    rep[f] Object g;                      Object g;

    void doTest(){                        void doTest(){
        f = new rep Object();                 try {
        g = new rep Object();                     handler.SetOwnerRep(this);
        doSth();                                  f = new rep Object();
    }                                             handler.SetOwnerRep(this);
                                                  g = new rep Object();
    void pure doSth(){}                           doSth();
}                                             } finally {
                                                  mergeOwnerSet(f,g);
peer T t = new T();                           }
t.doTest();                               }

                                          void doSth(){}
                                      }

                                      handler.SetOwnerPeer(this);
                                      T t = new T();
                                      t.doTest();
```

As the example shows an object of type T is created. Then doTest is executed, which calls doSth. But when callingdoSth the fields f and g do not refer into the same cluster. In order to be sure that f and g point into the same cluster the mergeOwnerSet has to be called before any method calls which are executed on peer objects (includes calls on this). It is not necessary to call mergeOwnerSet before calling methods on rep objects because the rep object has no access to caller object's fields. Calling methods on readonly objects is not allowed from the type system so the additional checks only have to be performed when calling methods on peer objects.

The class T has to perform an addition mergeOwnerSet call before calling doSth in doTest.

```
class T {
    Object f;
    Object g;
    void doTest(){
        try {
            handler.SetOwnerRep(this);
            f = new rep Object();
            handler.SetOwnerRep(this);
            g = new rep Object();
            mergeOwnerSet(f,g);
            doSth();
        } finally {
            mergeOwnerSet(f,g);
        }
    }

    void doSth(){}
}
handler.SetOwnerPeer(this);
T t = new T();
t.doTest();
```

4.1.2.5 Instanceof Expressions

The code transformation is the same as in Universe type system. The code transformation for rep and peer instanceof expression is as follows:

```
class T {
    uniq Object f;
    rep[f] Object g;

    void doSth(readonly Object x){

        if (x instanceof peer Object){


            f = new rep Object();

            g = new rep Object();

            doSth2(g);
        }

        doSth2(new rep Object());
    }



    void pure doSth2(readonly Object y){

        if (y instanceof rep[f] Object){

        }
    }
}



peer T t = new T();
t.doSth(t);
t.doSth(new rep T());
```

```
class T {
    Object f;
    Object g;

    void doSth(readonly Object x){
        try {
            if (x instanceof Object
              && handler.isPeer(x,this)){
                handler.SetOwnerRep(this);
                f = new Object();
                handler.SetOwnerRep(this);
                g = new Object();
                mergeOwnerSet(f,g);
                doSth2(g);
            } else {
                mergeOwnerSet(f,g);
                doSth2(new rep Object());
            }
        } finally {
            mergeOwnerSet(f,g);
        }
    }
    void doSth2(Object y){
        try {
            if (y instanceof Object
                    && handler.isPeer(y,f)){
            }
        } finally {
            mergeOwnerSet(f,g);
        }
    }
}
handler.SetOwnerPeer(this);
T t = new T();
t.doSth(t);
handler.SetOwnerRep (this);
T $t1 = new T();
t.doSth($t1);
```

As the example shows for a “x instanceof rep[f]” expression an additional check handler.isPeer(x,f) is performed to check if x and f refer into the same cluster. For a “x instanceof peer Object” the additional handler.isPeer(x,this) is needed to check whether x and this refer into the same cluster.

4.1.2.6 Cast Expressions

For cast expression the same additional method call handler.isPeer is added.

```
class T {
    uniq Object f;
    rep Object g;

    void doCast(readonly Object x){
        g = (rep[f] Object)x;
    }




    void doCast2(readonly Object x){
        peer Object p=(peer Object)x;
    }
}
```

```
// only code transformation for cast
expression is shown:

class T {
    Object f;
    Object g;

    void doCast(if (x Object x){
        if ((x instanceof Object
         && handler.isPeer(x,f))==false){
            throw new CastException());
        }
        g = x;
    }
    void doCast2(Object x){
        if ((x instanceof Object
         && handler.isPeer(x,this))==false){
            throw new CastException());
        }
        Object p = x;
    }
}
```

As the example shows cast expression as “`(peer Object)x`” are transformed into:

```
if((x instanceof Object && handler.isPeer(x,this))==false){
    throw new CastException();
}
```

Casts like “`(rep[f] Object)x`” are transformed into:

```
if((x instanceof Object && handler.isPeer(x,f))==false){
    throw new CastException();
}
```

handler.isPeer returns false if the one of the arguments (x or f) is null. A cast expression as
“`(rep Object)x`”
cannot be transformed because the cluster is not defined. The Uniqueness forbids cast expression as
“`(rep Object)x`”
where the cluster is not defined.

As already defined assignments like
“`peer Object x = new rep Object()`”
perform merge of cluster of left hand side with right hand side. If an Object declared as rep is used in a cast expression and the cast expression’s destination is peer then a merge operation is done first and then the tested object is in the right cluster and handler.isPeer returns true.

4.2 Generics

Before discussing the model of the runtime implementation for generic types it must be defined how the runtime type of a generic object has to be defined. Because this is a implementation specific topic it is discussed here and not in chapter three.

4.2.1 The Runtime Type of Generic Objects

In this subchapter the runtime type of generic objects is defined. As the runtime type needs a lot of aspects to keep in mind it is needed to describe them in the detail. Without this description it is hard to understand why the solution at the end is chosen.

4.2.1.1 Object Creation

The code example below shows in the method start the creation of a generic object which is assigned to x. The type arguments of x contain another type arguments.

```
public class Item<X>{}

public class Pair<Y,Z>{}

public class Test {

    public void start(){
    rep Pair<peer Pair<rep Integer, peer String>,
        rep Item<peer Integer>> x =
    new rep Pair<peer Pair<rep Integer, peer String>,
        rep Item<peer Integer>>();
    }
}
rep Test t = new Test();
t.start();
```

In the first step a structure has to be found where the type of x can be stored. The type of the local variable x can be represented as the following tree:

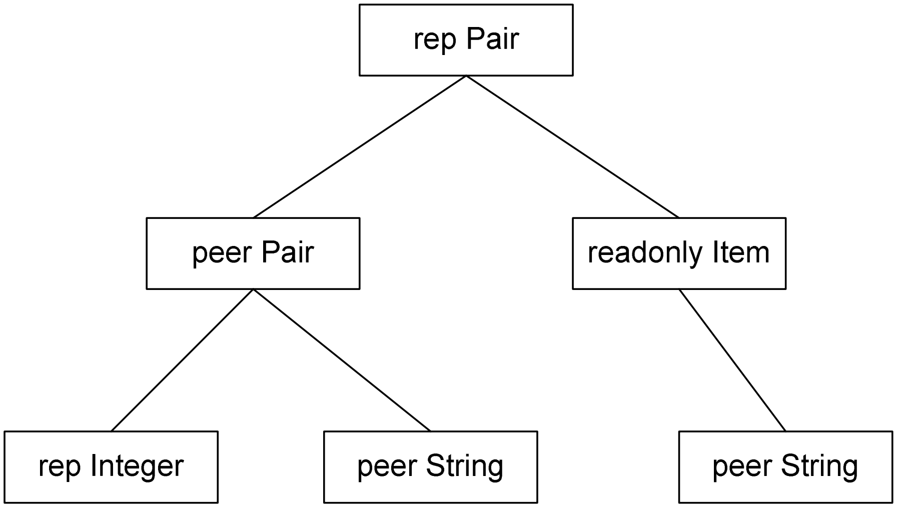


Figure 4-1: Tree Representation of a Generic Type

Figure 4-1 shows how the static type of x represented as a tree. The root node of the tree represents the main type of x. The modifier in every node is used for the static type checks. Every modifier is replaced by a reference to an object which represents the owner according to the modifier.

```
public class RuntimeType {
    Class classId;
    Object owner;
    RuntimeType[] children;
}
```

The class definition of RuntimType describes a possible solution for representing the runtime type of a generic. With RuntimeType the runtime type of x looks as follows:

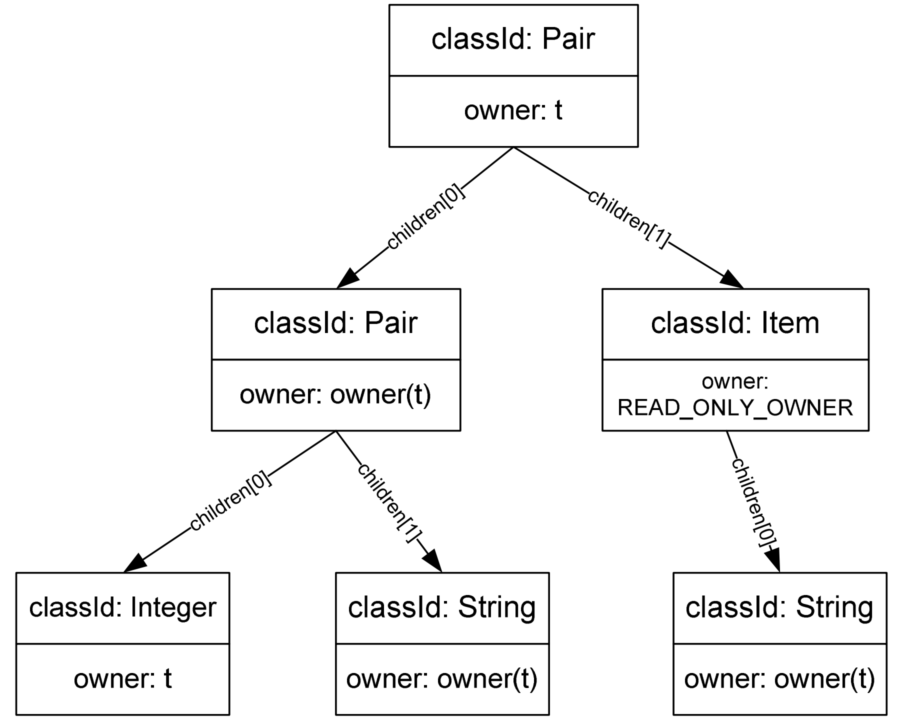


Figure 4-2: Runtime Type of a Generic Object

The runtime type of x shows all type arguments' owner and class identifier. owner(t) is the owner of the object referenced by t. This runtime model has the disadvantage that six objects of type RuntimeType have to be created. This representation will consume too much memory. Therefore RuntimeType is not used and it is tried to find another data structure which consumes less memory.

The first approach was to store the tree structure as an array of int. The length of this array is equal the number of nodes in the tree including the root node. The algorithm for storing the tree in a one dimensional tree is written in pseudo code and looks as follows:

```
// function
void parse(Node node) {
    while (pos<nofChildren){
        nofChildren[pos] = numberOfChildren(node);
        pos++;
        for all nodes=children(node) {
            parse(nodes[i]);
        }
    }
}

// start
int pos = 0;
int[] nofChildren = new int[#Nodes];
parse(rootNode);
```

description:

- pos is a counter which starts with 0.
- #Nodes is the total number of Nodes the tree has (including the root Node).
- nofChildren is the int array storing at each position the number of children a node has.
- numberOfChildren(node) returns the number of children a node has.
- children(node) returns all direct children of the Node node as an array.

The int[] for the tree structure in Figure 4-2 looks as follows:

$$nofChildren' = \{2, 2, 0, 0, 1, 0\}$$

With this representation a structure of the tree can be easily rebuilt.

The direct children of the root node represent the runtime type of type variables. Remember the class definition of

```
public class Pair<Y,Z>
```

and the runtime type shown in Figure 4-2 the runtime type of the type variable Y is the first child of the root node the one of Z is the second child of the root node. Creating objects as shown in the method start and described in “2.3.2 Generic Types in the Universe Type System” we need to look up the runtime type of Y and store for the Item it.

```
public class Item<X>{}

public class Pair<Y,Z>{
    public void start(){
        rep Item<Y> it = new rep Item<Y>;
    }
}
```

If we want to find out the runtime type of Y and we know that the tree structure of the runtime type of the object where start is called looks as

$$nofChildren' = \{2, 2, 0, 0, 1, 0\}$$

then another algorithm is needed to find where the runtime type of the type variable Y is stored . The structure of the tree of the runtime type of the type variable Y represented as an array is:

$$nofChildren_Y = \{2, 0, 0\}$$

To speed up the time for getting the runtime type's tree structure of type variables the integer array is replaced by a two dimensional array. The runtime type of the root node is stored already in the current runtime model [**7**]. Every direct child of the root node represents a type variable. As each direct child dc of the root node represents a tree again the nofChildren array is generated for each dc according the showed algorithm. Then the arrays for the runtime type of the object x look as follows:

$$nofChildren\ _Y = \ \{2, 0, 0\}$$

$$nofChildren\ _Z = \ \{1, 0\}$$

Now the two dimensional array nofChildren looks as follows:

$$\begin{aligned} nofChildren &= \{nofChildren_Y, nofChildren_Z\} \\ &= \{\{2, 0, 0\}, \{1, 0\}\} \end{aligned}$$

With this representation the type variable Y is at nofChildren[0] and Z is at nofChildren[1] and they can be found in constant time. This two dimensional array of int representation is used for storing the tree structure of the runtime type of objects.

Every entry in nofChildren stores the number of children of a certain node. Therefore this entry belongs to a node in the tree. Therefore the structure of nofChildren is reused to store the owners and class ids for each node.

```
Object[][] owners;
Class[][] classIds;
```

owners and classIds do have the same dimension as nofChildren. Instead of storing the number of children for each entry the reference to the owner (for owners) and the reference to the class identifier (for classIds) ared used instead.

For the tree structure of the runtime type of the type arguments of x:

```
int[][] nofChildren = {{2,0,0},{1,0}}
```

the corresponding owners and classIds look as follows:

```
owners = {{owner(t),t,owner(t)},{READ_ONLY_OWNER,owner(t)}}
classIds = {{Pair,Integer,String},{Item,String}}
```

The runtime type of the local variable is now stored in three two dimensional arrays nofChildren, owners and classIds. With this representation it is not needed to allocate an object for each type argument to store the runtime type of x.

4.2.1.2 Object Creation with Type Variables

In 4.2.1.1 the solution for representing the runtime type for a generic object is

```
int[][] nofChildren;
Object[][] owners;
Class[][] classIds;
```

where every int in nofChildren denotes the number of children the corresponding type argument has. The next code example shows the creation of a generic object which uses class type variables as type arguments.

```
public class XPair<X,Y> {

    void createItem(){
        Item<X> item = new Item<X>();
    }
}
rep XPair<rep Integer, rep String> x =
      new rep XPair<rep Integer, rep String>();
x.createItem();
```

The runtime type for x is:

```
nofChildren = {{0},{0}};
owners = {{x},{x}};
classIds = {{Integer},{String}};
```

The runtime type for item must be looked up at the runtime type of x. The runtime type for the type variable X can be looked up at runtime where the runtime type for x is stored. There X is stored at nofChildren[0], owners[0] and classIds[0](because X is list at the first position in the class definition of XPair). Somehow the index has to be stored where the runtime type of the type variable X of x can be found. The easiest way (we found) without changing the runtime representation was to modify the nofChildren. If the runtime type is a type variable as the X in new Item<X> we treat this type variable as a non generic type but store in nofChildren a negative value which describes the index where to find the runtime type. If an entry in nofChildren is negative, we know that this entry denotes a type variable. The runtime type of this type variable can be looked up at the runtime type of the object which uses this type variable to create the object. In the code example above where the method createItem on x is called the type variable X for creating item can be looked up at the runtime type of x.

The described example uses class type variables. But objects can be created with method type variables too as the following code example shows:

```
public class XPair<X,Y> {

    <K,V> void doSth(K k, V v){
        rep Item<K> item = new rep Item<K>();
    }
}
```

The problem is now that for creating item in the method doSth the type variable cannot be looked up in the object on which doSth is called. The runtime type for the method type variables is always stored on a global defined stack before calling the method. The method type variables are indicated with a negative value in nofChildren too and they have to be looked up at the global defined stack.

As already mentioned the negative values in the elements of nofChildren denote indexes for class type variables or method type variables. We decided to define negative odd values to indicate class type variables and even negative values for method type values.

We define the function $classTypeIndex$ for the type variables in a class definition as:

```
class G<X1, X2, …, Xi, … Xn>{}
```

$$classTypeIndex(X1) = -1$$

$$classTypeIndex(Xi) = -(2 * i) + 1$$

and the function $methodTypeIndex$ for the one for method type variables is defined as:

```
<K1, K2, …, Ki, … Km> void aMethod(K1 k…){}
```

$$classTypeIndex(K1) = -2$$

$$classTypeIndex(Xi) = -(2 * i)$$

The class org.multijava.universes.rt.impl.generics.GenericsEncoding provides static methods for creating negative values for class or method type variables and vice versa.

4.2.1.3 Multijava and Object Creations

Before considering the interaction between multijava and the runtime representation of generic objects the representation is shown again:

Every generic object has a runtime type. The runtime types of the type arguments are represented as

```
int[][] nofChildren;
Object[][] owners;
Class[][] classIds;
```

where a negative value in an element in nofChildren indicates that a type variable has to be replaced by the actual runtime type. As the code below shows:

```
rep Item<F> item = new rep Item<F>
```

If the compiler finds an object creation expression as on the right hand side of the assignment in the code above the compiler adds a method call expression which is responsible to store the runtime type of item. All the method calls which the compiler generates for the runtime support for generic types are defined in the interface TypeArgumentsHandler. For the following methods the compiler creates method calls. For every method it is described when it is used.

Note: gH is used as shorthand which points to an instance of TypeArgumentsHandler.

If the compiler finds an object creation expression the compiler generates the method call

```
-   void addTypeArguments(Object current, int[][] nofChildren,
    Object[][] owner, Class[][] classIds);
```

or

```
-   void addTypeVariables(Object current, int[][] nofChildren,
    Object[][] owner, Class[][] classIds);
```

For an object creation expression which is shown in the code below for the local variable item where no type variables are used

```
public class Item<K>{}

public class T<P> {
    void createItem(){
        rep Item<rep Integer> item = new rep Item<rep Integer>();
    }
}
```

the following code transformation is generated:

```
public class T<P> {
    void createItem(){
        gH.addTypeArguments(this,{{0}},{{this}},{{Integer}});
        Item<Integer> item = new Item<Integer>();
    }
}
```

The method addTypeArguments can store the parameters without any other restrictions.

Consider now the next example with an object creation expression containing type variables as follows:

```
public class Item<K>{}

public class T<P> {
    void createItem(){
        rep Item<P> item = new rep Item<P>();
    }
}
```

The code transformation looks as follows:

```
public class T<P> {
    void createItem(){
        gH.addTypeVariables(this,{{-1}},{{}},{{}});
        Item<P> item = new Item<P>();
    }
}
```

in the method addTypeVariables a lookup on the runtime type of this is performed to replace the runtime type of the type variable P. This runtime type P is copied into

```
{{-1}},{{}},{{}}
```

Afterwards nofChildren, owners and classIds are:

```
{{0}},{{this}},{{Integer}}
```

So the method addTypeVariable has to traverse the parameter nofChildren to find negative values. If it finds negative values the lookup, copy and replacement of the type variable has to be performed. To save the time for the traversal of nofChildren if no type variables are used the method addTypeArguments is called which does not perform the traversal.

When the methods addTypeArguments and addTypeVariables are called as shown in the code examples above the object referenced by item is not assigned to its runtime type yet. Therefore in the constructor of the class Item the additional method call

- `void registerTypeArguments(Object current);`

is needed. Every class which uses class type variables includes registerTypeArguments as the first statement (or the second statement, if super() is the first statement) in its constructor(s) as shown in the code below for the default constructor:

```
public class Item<K>{
    public Item() {
        gH.registerTypeArguments(this);
    }
}
```

The reason for this additional call in the constructor is the same as already discussed in Runtime Checks for the Universe Type System [7]. In the constructor of Item a new object could be created which uses already the runtime type of K. Therefore the runtime type of the new object must be assigned to the new object first.

The runtime type of class type variables can always be looked up at the runtime type of **this**. But if the type variables are method type variables the runtime type of the type variable has to be stored separately on a global defined stack. The stack can be accessed by the methods

-
  ```
  void registerMethodTypeVariables(Object current,
  int[][] nofChildren, Object[][] owners, Class[][] classIds);
  ```

and

- `void unregisterMethodTypeVariables();`

The next code example shows a method call which uses method type variables:

```
void test(){
    rep Integer i = new rep Integer(1);
    peer String s = "aString";
    doSth(i,s);
}

<K,V> void doSth(K k, V v){
    rep Item<K> it = new rep Item<K>();
}
```

before the execution of doSth in the method test the method call registerMethodTypeVariables have to be performed. After the execution of doSth the method call unregisterMethodTypeVariables has to be performed too. Because of exception handling the latter method call has to be performed in any cases. Therefore the two methods and the method call doSth are packed into a try finally statement as the code transformation below shows:

```
void test(){
    rep Integer i = new rep Integer(1);
    peer String s = "aString";
    try {
          gH.registerMethodTypeVariables(this,
                {{0},{0}},{{this},{owner(this)}},{{Integer},{String}});
        doSth(i,s);
    } finally {
        gH.unregisterMethodTypeVariables(this);
    }
}
```

With the runtime types of the method type variables K and V a type arguments representation is created first which looks as K and V were used in a new object expression as where G is only a place holder for a class name which is not further used:

```
new G<K,V>
```

As already written in the code transformation above the representation of the runtime of G is:

```
nofChildren = {{0},{0}}
owners = {{this},{owner(this)}}
classIds = {{Integer},{String}}
```

The method call

```
-   boolean checkCast(Object src, int[][] nofChildren,
    Object[][] owner, Class[][] classIds);
```

is added if a cast looks as follows:

```
(rep Item<rep Integer>) x
```

The method call

```
-   boolean checkCastWithTypeVar(Object src, Object current,
    int[][] nofChildren, Object[][] owner, Class[][] classIds);
```

is added if the cast includes type variables as shown in the following expression

```
(rep Item<K>) x
```

The method call

```
-   boolean checkInstanceOf(Object src, int[][] nofChildren, Object[][]
    owner, Class[][] classIds);
```

is added if the instanceof includes looks as follows:

```
x instanceof rep Item<rep String>
```

And the method call

```
-   boolean checkInstanceOfWithTypeVar(Object src, Object current,
    int[][] nofChildren, Object[][] owners, Class[][] classIds);
```

is added if the cast includes type variables as shown in the following expression

```
x instanceof rep Item<K>
```

For the cast and instanceof expressions the parameters nofChildrenObj, owners and classIds are created from the destination of the instanceof and cast expressions. The parameter src is x and current is **this**.

4.2.2 Concept

The main idea of the runtime support for generic types is to provide an implementetation for the Universe Types and one for Uniqueness. Then it must be possible to decide at runtime how the type arguments information has to be stored and checked. To fulfill these requirements the following model is used.

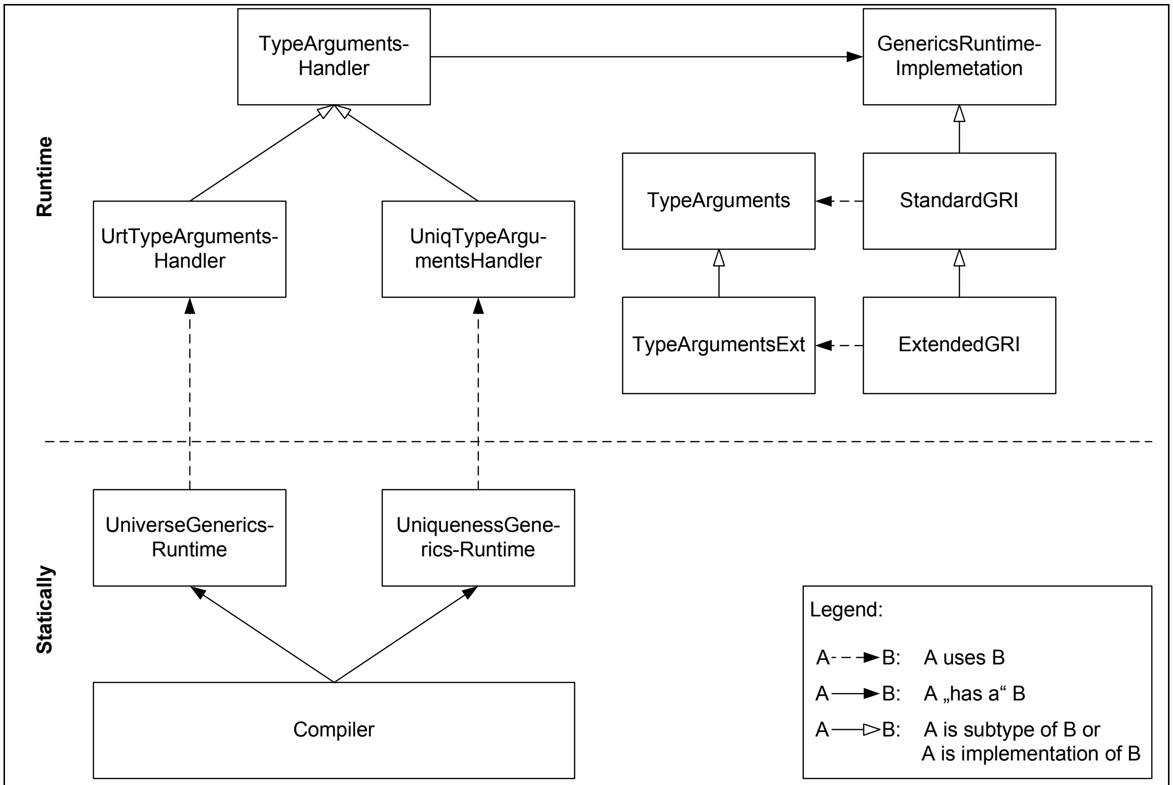


Figure 4-3: Class Model for the Generics Implementation

The class TypeArguments stores the runtime type of generic objects. It stores nofChildren and owners of the type arguments. The class TypeArgumentsExt is a subtype of TypeArguments and stores the class identifiers of the type arguments too. The GenericsRuntimeImplementation is an interface for creating TypeArguments checking TypeArguments as this is needed for instanceof or cast expressions. The StandardGRI an implementation of GenericRuntimeImplementation creates and checks TypeArguments where ExtendedGRI creates and checks TypeArgumentsExt. The difference between StandardGRI and ExtendedGRI is that the latter includes creating and checking the class identifiers. The compiler depending if Uniqueness is enabled or not has a UniverseGenericsRuntime or a UniquenessGenericsRuntime object. These two objects support the compiler to create the method calls to a TypeArgumentsHandler for maintaining the runtime type for generic objects. UrtTypeArgumentsHandler is used if the compiler flag for Uniqueness is off. UniqTypeArgumentsHandler is used if the compiler flag for Uniqueness is on. At runtime depending on the Java system property “UniverseGenericsImplementation” the TypeArgumentsHandler has a corresponding reference to a GenericsRuntimeImplementation. The programmer can implement its own GenericsRuntimeImplementation and can use it by setting the system property as

```
java – UniverseGenericsImplementation =org.multijava.universes.rt.impl.generics.ExtendedGRI …
```

If the system property UniverseGenericsImplementation is not set the StandardGRI is used. The TypeArgumentsHandler lets the GenericsRuntimeImplementation creating the TypeArguments. These TypeArguments are stored by the TypeArgumentsHandler in a hash table. If a method call as `checkInstanceOf` as already described in “4.2.1.3 Multijava and Object Creations” the TypeArgumentsHandler gets the stored TypeArguments for the object which has to be tested and calls the corresponding method on GenericsRuntimeImplementation which evaluates the instanceof expression.

4.2.3 Modified and New Packages

The **org.multijava.universes.rt** package is extended with the following interfaces:

- GenericsSupport: interface for supporting generics in UrtHashtable.
- UniverseGenericsRuntime: reuse of the UniverseRuntime, contains a static initializer which loads an implementation of the UniverseGenericsImplementation. The default implementation is StandardGRI (described later). The loaded implementation is used by the TypeArgumentsHandler (described later).
  The user can use other UniverseGenericsImplementation by using java system property:
  java –D UniverseGenericsImplementation=ExtendedGRI
- UniquenessGenericsRuntime: reuse of the UniverseGenericsRuntime for providing runtime support for Uniqueness.

The **org.multijava.universes.rt.impl.generics** is created and contains the following classes:

- GenericsEncoding: provides generating negative values for denoting class and method type variable indexes
- GenericsHashtable: reuse of UrtHashtable, for managing the generic objects’ type arguments.
- GenericsRuntimeImplementation: Interface for checking type arguments for cast and instanceof expression and for creating TypeArguments (described later).
- StandardGRI: A GenericRuntimeImplementation implementation which creates and checks TypeArguments only owner entries.
- ExtendedGRI: A GenericsRuntimeImplementation implementation which creates and checks TypeArguments owner and class identifier entries.
- TypeArguments: for storing type arguments, only stores owners of type arguments.
- TypeArgumentsExt: Extends TypeArguments and stores class identifiers of type arguments too.

- TypeArgumentsHandler: interface for defining the operations for the runtime support for generic types.
- UrtTypeArgumentsHandler: Implementation of TypeArgumentsHandler, can be accessed over org.multijava.universes.rt.UniverseGenericsRuntime.handler. Manages type arguments of objects and stores item in GenericsHashtable.
- UniqTypeArgumentsHandler: reuse of UrtTypeArgumentsHandler for Uniqueness.

The statical type checks for generic types are not defined in Uniqueness yet. The runtime support for generics in Universe type system is developed with respect to Uniqueness. Once the statical Type Checks for Uniqueness are defined, the classes

- UniquenessGenericsRuntime
- UniqTypeArgumentsHandler

have to be implemented. UniqTypeArgumentsHandler is a subclass of UrtTypeArgumentsHandler where most of the implementation can be reused. Some Uniqueness specific implementations are already written in UniqTypeArgumentsHandler and UniquenessGenericsRuntime by this project.

4.2.4 Source Code Transformation

Source code transformation for generic types is needed by creating new objects, instanceof checks and casts. In this chapter only the code transformation is shown which is depending on generic types. For overview reasons we use

gH=org.multijava.universes.rt. UniverseGenericsRuntime.handler

as an abbreviation.

4.2.4.1 Object Creation

Most of the code transformation is already described in "4.2.1.3 Multijava and Object Creations". Therefore it is not necessary to discuss it again in detail but the most important steps are explained again. Every constructor of a generic class needs the method call to registerTypeArguments. If a new object expression uses type variables the method call addTypeVariables is inserted. If no type variables are used the method call addTypeArguments is inserted.

class Node<K,V>{}	class Node<K,V>{
	Node<K,V>(){
	gH.registerTypeArguments(this);
	}
	}
class List<K,V>{	class List<K,V>{
	List<K,V>(){
	gH.registerTypeArguments(this);
	}
rep Node<K,V> first;	Node<K,V> first;
void init(){	void init(){
first = new rep Node<K,V>();	*gH.addTypeVariables(*
}	*{{-1},{-2}},*
}	*{{ },{}},*
	{{},{ }});
	first = new Node<K,V>();
	}
	}
peer List<rep Integer,	*gH.addTypeArguments(*
peer String> l = new peer	*{{0},{0}},*
List<rep Integer,	*{{this},{owner(this)}},*
peer String>();	*{{Integer},{String}});*
	List< Integer, String> l = new List<
	Integer, String>();

4.2.4.2 Instanceof Expressions

For instanceof expression an additional check by calling gH.checkInstanceOf is needed.

class Node<K,V>{	class Node<K,V>{
void foo(readonly Object x){	void foo(readonly Object x){
if (x instanceof peer Node<rep	if (x instanceof Node *&&*
Integer, repString>) {}	*gH.checkInstanceOf(x,*
}	*{{0},{0}},*
}	*{{this},{this}},*
	{{Integer},{String}})){}
	}
	}

For an instanceof check the destination type is represented as the three two dimensional arrays and then passed to the checkInstanceOf method which compares the arrays with the runtime type of x. If an instanceof expression contains type variables gH.checkInstanceOfWithTypeVar is called which replaces first the type variables by its runtime type.

4.2.4.3 Cast Expressions

For cast expression an additional check by calling gH.checkCast is needed.

```
class Node<K,V>{
    void foo(readonly Object x){
        peer Node<rep Integer, repString> n =
            (peer Node<rep Integer, repString>)x;
    }
}
```

```
class Node<K,V>{
    void foo(readonly Object x){
        if ((x instanceof Node &&
            gH.checkCast(
                this,
                {{0},{0}},
               {{this},{this}},
               {{Integer},{String}}))==false)
        {
                throw new CastException());
        }
        n = x;
    }
}
```

For a cast check the destination type is represented as the three two dimensional arrays and then passed to the checkCast method which compares the arrays with the runtime type of x. If a cast expression contains type variables gH.checkCastWithTypeVar is called which replaces first the type variables by its runtime type.

4.3 Benchmarks

Testing the performance of the new runtime implementations for Uniqueness and for Generic Types is a big topic. As the current runtime support [**7**] implemented by Daniel Schregenberger already lower the performance distinctively we attempt that the performance for the runtime support for Uniqueness can be measured relatively to the current runtime support. For the runtime support for generics it will be very interesting to measure the performance and to relate it with the current runtime support. With this project a framework is built which allows the programmer to easily benchmark implementations. Unfortunately the time for this project was too short to evaluate good benchmark examples.
All the needed files for the benchmarks as well as some possible java files to test the performance is stored in the folder Benchmarks on the CD provided by this project.
Provided java Files:

- Adjacency: Produces a given number of nodes. Decides randomly if two nodes have to be linked together.
- AnonymousRep/AnonymousPeer: creates anonymous rep/peer objects
- BinaryTree: Creates a binary tree of a given height. For each node every child is created randomly.
- DivideAndAppend: A kind of quick sort algorithm. Links the entries together instead of sorting them.
- Queue: Puts a given number of elements into a queue with a given capacity.

4.3.1 Framework

All the tests are measured with the linux command time. The shell script iterator.sh provides two functions setJava and iterate.

```
alias jUniq='java org/multijava/mjc/Main --universesx=parse,check,purity,xbytecode,annotations,uniq'
alias jUniqRt='java org/multijava/mjc/Main --universes'
alias jUrt='java org/multijava/mjc/Main --universesx=parse,check,purity,xbytecode,annotations,dynchecks'




function setJava {
  runJava="$JAVA $1"
  #Echo "runJava is set to "$runJava
}




#iterate(what, from, to, numberOfRepetitions, steps, whereToStore)
function iterate {
  elems=$2
  while ((elems<=$3)); do
    a=$4
    while ((${a}>0)); do
    /usr/bin/time -v --output=t.txt $runJava $1 ${elems};
    grep "User time" t.txt | tee -a results/${6}.yresults;
    grep "System time" t.txt | tee -a results/${6}.zresults;
    grep "Command being timed" t.txt | tee -a results/${6}.xresults;
    a=${a}-1;
    done
    elems=$((elems+$5));
  done
}
```

The alias jUniq, jUniqRt and jUrt are shorthands for compiling the java files depending on whether Uniqueness is on or off and runtime support is enabled or not.
jUniq defines that a Java file is compiled with Uniqueness enabled and runtime support disabled.
jUniqRt defines that a Java file is compiled with Uniqueness enabled and runtime support enabled.
jUrt: defines that the Java file is compiled with Uniqueness disabled and runtime support enabled.

The function setJava allows the user to add flags to the $JAVA environment variable for example system properties.

1. The function iterate executes the class file “what” “numberOfRepetitions” times with the parameter “from”. Then the result is stored in the folder result on three files with the names “what”.xresults, “what”.yresults and “what”.zresults. The xresults file contains the command used for the execution of the java file as f.e.:
   $JAVA AnonymousRep 10000
2. The yresults store the execution time in user mode for this task.
3. The zresults store the execution time in system mode for this task.
4. For each execution of the class file the entries in the xresult, yresults and zresults are appended.
5. The parameter “from” is increased by “steps”. If from is smaller or equal “togo to 1.

At the end for each executed class file there are three results files in the folder results having the same name as the executed class file and ending with xresults, yresults and zresults.

Some possible test cases are already provided. To show the functionality the file TestBinaryTree.sh is described:

```
start=10
end=20
rep=5
steps=1

#iterate(what, from, to, numberOfRepetitions, steps, whereToStore)

jUniq BinaryTree.java
iterate BinaryTree $start $end $rep $steps BinaryTree

jUniqRt BinaryTree.java
iterate BinaryTree $start $end $rep $steps BinaryTreeUniqRt

jUrt BinaryTree.javaOld
iterate BinaryTree $start $end $rep $steps BinaryTreeUrt
```

This script file calls first jUniq BinaryTree.java which compiles the file BinaryTree.java. Then iterate is called.

The file Test.sh is the main:

```
. iterator.sh
touch t.txt

mkdir -p results
rm -f -r old
mkdir old
cd results
mv * ../old
cd ..

setJava

. TestAnonymousPeer.sh
. TestAnonymousRep.sh
. TestBinaryTree.sh
. TestDivideAndAppend.sh
. TestQueue.sh
```

Test.sh load the founction setJava and iterate from iterator.sh, moves the content from the folder result into old. The function setJava is called without parameter because here no system property for $JAVA is used. Then the files like TestBinaryTree.sh are called. At the end all the results of all executet files as TestBinaryTree.sh are stored in the folder results.

4.4 Problems with Multijava

There are some problems which could not have been solved.

4.4.1 Creating Bytecode for Int Arrays

Some code transformations which are performed by the runtime support for generic types und Uniqueness need to create arguments of type int[][]. This is used for example by generating method calls to org.multijava.universes.rt. UniverseGenericsRuntime.handler.addTypeArguments(Object current, int[][] nofChildren, Object[][] owners, Class[][] classIds). I could not manage creating the argument nofChildren with the type nofChildren. In other word the method call I wanted to create as

```
…addTypeArguments(this,
    int[][] {int[]{0}},
    Object[][]{Object[]{this}},
    Class[][]{Class[]{Integer}})
```

failed because the type declaration int[][] was not accepted. Therefore I replaced it by Object and used

```
…addTypeArguments(this,
    int[][] {int[]{0}},
    Object[][]{Object[]{this}},
    Class[][]{Class[]{Integer}})
```

instead which works without any problems. The signature for addTypeArguments still contains type declaration int[][] for the second parameter.

4.4.2 Method Type Variables Information Losses

The JMethodCallExpression provides a HashMap hash in the method typecheck. hash provides the mapping from method type variables to actual the actual type these type variables are assigned. If the user wants to compile a file as:

```
public class foo {
    public void bar () {
        rep C<rep Integer, peer String> x =
        new rep C<rep Integer, peer String>();
        peer C<peer Integer, readonly String> y =
        new peer C<peer Integer, readonly String>();
        doSth(x,y);
    }

    <K,V> void doSth(K k, V v){
        System.out.println(k instanceof rep C<rep Integer, peer String>);
    }

    public class C<A,B>{}
}
```

No errors or warnings are generated. But if the user changes the method call from `doSth(x,y)` to `this.doSth(x,y)`. The multijava hangs up because of a NullPointerException which is caused by hash. I disabled operation at this position if a NullPointerException would occur and create a CUniverseMessages message which returns an error message to the user.

4.5 Universe Messages

Two new messages are added:

The error message described in 4.4.2 is

- `METHOD_CALL_WITH_METHOD_TYPE_VARIABLES_ON_THIS`

type arguments which are raw types cannot be checked at runtime. Therefore the following warning is produced:

- `RAW_TYPE_FOUND_IN_GENERIC_TYPE`

4.6 Test Cases

For the runtime support for Uniqueness and generic types two new folders with test cases are added:

- org/multijava/mjc/testcases/universes/runtime/uniqueness:
  It covers all related testcases for testing the runtime support for Uniqueness without generics.
- org/multijava/mjc/testcases/universes/runtime/generics:
  It covers all related testcases for thesting the runtime support for Generics in the Universe Types. For Uniqueness no testcases are provided because generic types are not supported by the statical type system yet.

In both folders the test cases can be started with the command “make runtests”. make runtests compiles every java, executes every class file and checks whether the output produced is expected or not. If it is not expected a message is printed out.

5 Conclusion and Future Work

The Universe type system now supports runtime checks for generic types. As for the runtime checks for non generic types where the owner of objects is stored the concept of storing the owner of the type arguments of generic types are reused. The user has the possibility to let the system check the class identifiers of the type arguments too. The user does not have to decide this before compiling time. He can decide this before runtime by setting a Java system property.

The Uniqueness and ownership transfer support now runtime checks for non generic types. The concept and most of the implementation of the basic runtime checks for the Universe type system are reused for the runtime checks for Uniqueness and ownership transfer. Owners of objects are now modeled with cluster objects. Clusters are sets of objects and are merged and transferred by the set union find structure. The static type checks for generic types in Uniqueness and ownership transfer are not implemented yet. Hence the runtime support for generic types in Uniqueness and ownership transfer could not be implemented completely. The runtime support for the Universe type system is implemented in a way that it can be easily reused for implementing the runtime support for Uniqueness and ownership transfer. A possible implementation which manages the runtime types for generic objects for Uniqueness and ownership transfer is already implemented by this project.

5.1 Future Work

- **Runtime support for array types in Uniqueness**:
  The runtime support for array types in Uniqueness and ownership transfer is not implemented yet. When the static type checks are implemented the runtime support for Uniqueness and ownership transfer can be extended.
- **Modification of the current runtime support for array types in the Universe type system**:
  With the newly implemented static checks for generic types the model for array types has changed. As an array has two modifiers the viewpoint of the second modifier is not relative to the first modifier anymore. It is in the same viewpoint as the first modifier. The current runtime checks do not implement this change yet.
- **Implementation of generic types in Uniqueness and ownership transfer**:
  When the static type checks for Uniqueness and ownership transfer are extended to support generic types the runtime support which was implemented by this project can be completed. Especially the code transformation in the abstract syntax tree has to be implemented.
- **Performance checks of the runtime support in Uniqueness and ownership transfer**:
  With the provided framework
- **Performance checks of the runtime support for generic types in the Universe type system**:
  With the provided framework

Acknowledgement: The reader would like to thank Arsenii Rudich, Werner M. Dietl, Peter Müller and the entire sct group for the great support and the great time.

Bibliography

[1]	W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology (JOT)*, 4(8):5–32, October 2005.
[2]	W. Dietl and S. Drossopoulou and P. Müller. Generic Universe Types. *Foundations and Developments of Object-Oriented Languages (FOOL/WOOD '07),* 2007.
[3]	W. Dietl and S. Drossopoulou and P. Müller. Generic Universe Types. *European Conference on Object-Oriented Programming (ECOOP),* 2007. To appear.
[4]	Robin Züger. Generic Universe Types in JML. Master's thesis, ETH Zurich, 2007.
[5]	P. Müller and A. Rudich: Ownership Transfer in Universe Types *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA),* 2007. To appear.
[6]	Yoshimi Takano. Implementing Uniqueness and Ownership Transfer in the Universe Type System. Master's thesis, ETH Zurich, 2007.
[7]	Daniel Schregenberger, Runtime Checks for the Universe Type System, Semester Project, ETH Zurich, 2004
[8]	Gilad Bracha. *Generics in the Java Programming Language*. Sun Microsystems, http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf, 2004.
[9]	S. Alstrup, I. L. Gørtz, T. Rauhe, M. Thorup, and U. Zwick. Union-find with constant time deletions. In *Proc. 32nd International Colloquium on Automata, Languages and Programming (ICALP 2005). Lecture Notes in Computer Science*, volume 3580, pages 78–89. Springer-Verlag, jul 2005.
[10]	Curtis Clifton. MultiJava: Design, implementation, and evaluation of a Java-compatible language supporting modular open classes and symmetric multiple dispatch. Technical Report 01-10, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, November 2001. Available from archives.cs.iastate.edu.
[11]	Curtis Clifton, Todd Millstein, Gary T. Leavens, and Craig Chambers. MultiJava: Design rationale, compiler implementation, and applications. *ACM Transactions on Programming Languages and Systems*, 28(3), May 2006.

A Changes on the Abstract Syntax Tree

A.1 Changed Files for Uniqueness and Ownership Transfer

All the listed changes are done in the org.multijava.mjc package:

1. CUniverseRuntime:
   New class to load the right runtime helper. Loads CUniquenessRuntimeHelper if Uniqueness is enabled. Loads CUniverseRuntimeHelper if Uniqueness is disabled.
2. CUniverseRuntimeHelper:
   Before all methods were statical. Now they are put into a singleton which can be accessed over CUniverseRuntime.
3. CUniquenessRuntime:
   The same as CUniverseRuntimeHelper but provides methods for Uniqueness.
4. JAssignmentExpression:
   If the assignment must perform merging clusters then a JMethodCallExpression is added which performs the needed operation.
5. JBlock:
   The body of a JBlock can be null. This can be produced in a JTryFinallyStatements' finally clause.
6. JCastExpression:
   The JCastExpression produced already an additional JMethodCallExpression to check the owner of the source of the expression with the one of the destination. As cast expressions can now merge clusters some small modifications are needed to support the new operations.
7. JClassFieldExpression:
   Added a method for getting the universe of the prefix of the class field expression. This is used for JAssignmentExpressions.
8. JConstructorBlock:
   Every constructor block is put into the try clause of a JTryFinallyStatement. In the finally clause JMethodCallExpressions are inserted which merge the class fields which have to be in the same cluster.
9. JInstanceofExpression:
   The same procedure is performed as described in JCastExpression.
10. JMethodCallExpression:
    If the passed parameter perform an ownership transfer (f.e. if the universe of the argument is rep and the universe of the parameter is of type peer) additional method calls are added for merging clusters.
    If the JMethodCallExpression is a call on this or on a peer object an array of JMethodCallExpression is added which guarantees that all class fields are in the same cluster which are declared to be in the same cluster.

11. JMethodDeclaration:
    If the method not pure then its body is put into the try clause of a JTryFinallyStatement. Then in the finally clause JMethodCallExpressions are inserted which merge the class fields which have to be in the same cluster.
12. JNewArrayExpression:
    The runtime support for Uniqueness produces JNewArrayExpressions. They must not be checked by the runtime support. Therefore an additional check if universe is enabled is needed to disable the runtime support for array types if needed.
13. JNewObjectExpression:
    A flag is inserted which is needed if the JNewObjectExpression is on the right hand side of JAssignmentExpression.
14. JTryFinallyStatement:
    If the JTryFinallyStatement is created by the runtime support for Uniqueness it checks whether the method calls for merging the class fields are needed or need. If they are needed an array of JMethodCallExpression is added to the finally clause.

A.2 Changed Files for Generic Types in the Universe Type System

All the listed changes are done in the org.multijava.mjc package:

1. CUniverseGenericsRuntimeHelper:
   Provides helper functions for the runtime support for generics.
2. JCastExpression:
   An additional JMethodCallExpression is added for checking the type arguments.
3. JConstructorBlock:
   An additional JMethodCallExpression is added register the current object's type arguments. This is the pendant to the "setOwner" method call which is already produced by the basic runtime support.
4. JInstanceofExpression:
   An additional JMethodCallExpression is added for checking the type arguments.
5. JMethodCallExpression:
   If the MethodCallExpression contains method with type variables then a JMethodCallExpression which registers the runtime type of the method type variables is added before. These two calls are packed into a try clause of a JTryFinallyStatement. In the finally clause of this JTryFinallyStatement a JMethodCallExpression is added which unregisters the runtime type of the method type variables.
6. JNewObjectExpression:
   Adds a JMethodCallExpression to register the runtime types of the type arguments.

B Project Delivery

With this project a CD is delivered containing the following materials:

1. This project report
2. The whole multijava project with the implementation of the runtime support for Uniqueness and ownership transfer including the test cases. The implementation for the runtime support for generic types is included in this project. But as this version does not support Universe generic types it cannot be tested. The implementation is built on the delivery of Yoshimi Takano's master's thesis.
3. The whole multijava project with the implementation of the runtime support for generic types. All the test cases for the runtime support for generic types are included. This multijava project is built on the delivery of Robin Zueger's master's thesis. This version does include neither static nor runtime checks for Uniqueness and ownership transfer.
4. The whole implementation is done under Windows XP on an Eclipse environment with cygwin to compile the multijava project. As the compilation needs some tricks under cygwin all the scripts which are needed to set the environment variables correctly are added to both multijava projects.