Inferring Universe Annotations on the Presence of Ownership Transfer

Annetta Schaad

Master Project Report

Software Component Technology Group
Department of Computer Science
ETH Zurich

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

May - October 2007

Supervised by:
Arsenii Rudich
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 uses ownership to structure the object store and to control aliasing and modification of objects. In former work ownership transfer - which allows objects to migrate from one owner to another - was integrated in the Universe type system. It is based on the concepts of external uniqueness and alias burying. This report presents different extensions to the Universe type system with ownership transfer. The main extension concerns the inference of the ownership modifier for local variables. Further we apply the concept of subclass separation to the object store. We additionally introduce a solution for array handling that is derived from field access. As a last extension we weaken the external uniqueness invariant to allow passing multiple references on one transferred cluster via method invocation.

Contents

1 Introduction **9**
1.1 Background . . . 9
1.1.1 Ownership Type Systems . . . 9
1.1.2 Universe Type System . . . 9
1.1.3 External Uniqueness . . . 10
1.1.4 Alias Burying . . . 10
1.2 Universe Types with Transfer . . . 10
1.2.1 Cluster Transfer . . . 12
1.2.2 Data Flow Analysis . . . 13
1.2.3 Example . . . 13
1.3 JML and MultiJava . . . 17
1.3.1 JML . . . 17
1.3.2 MultiJava . . . 17
1.4 Project Goals . . . 17
1.5 Overview . . . 17

2 Type Inference for Local Variables **19**
2.1 Introduction . . . 19
2.2 Solution . . . 20
2.2.1 Local Variables . . . 20
2.2.2 Fields . . . 20
2.2.3 Cluster Transfer . . . 21
2.2.4 Cluster of New Variables . . . 21
2.2.5 Cast . . . 23
2.2.6 Input Parameters . . . 24
2.2.7 Return Values . . . 24
2.2.8 Static Methods . . . 24
2.2.9 Purity Checking . . . 24
2.2.10 free Variables . . . 25
2.2.11 Releasing Problem . . . 28
2.2.12 Summary . . . 29

3 Subclass Separation **31**
3.1 Introduction . . . 31
3.2 Solution . . . 31
3.2.1 Subclass Separation . . . 31
3.2.2 Notation . . . 32
3.2.3 Cluster for the Analysis . . . 32
3.2.4 Changes in the Semantic . . . 32
3.2.5 Summary . . . 33

4 Arrays **35**
4.1 Introduction . . . 35
4.2 Solution . . . 35
4.2.1 Idea . . . 35
4.2.2 Array Declaration . . . 36
4.2.3 Assignable-to Relation . . . 36
4.2.4 Array Object . . . 36
4.2.5 Array Elements . . . 37
4.2.6 Flattening . . . 38
4.2.7 Array Reading . . . 39
4.2.8 Array Writing . . . 39
4.2.9 Unusable Arrays . . . 40
4.2.10 Array Creation Expression . . . 41
4.2.11 Array Initializer . . . 42
4.2.12 Array Transfer . . . 43
4.2.13 Summary . . . 43

5 Passing Multiple References on a Free Cluster **45**
5.1 Introduction . . . 45
5.2 Solution . . . 46
5.2.1 Declaration Syntax . . . 46
5.2.2 Semantic . . . 46
5.2.3 Order of Type Checking and Transition Rules . . . 47
5.2.4 Summary . . . 52

6 Formalization **53**
6.1 Toy Language Syntax . . . 53
6.2 Clusters . . . 54
6.3 Universe Modifier Translation . . . 56
6.4 Lookup Functions . . . 57
6.4.1 Type Projection Functions . . . 58
6.5 Type Combinator . . . 58
6.6 Assignable-To Relations . . . 58
6.7 Ternary Logic . . . 60
6.8 Static Data Flow Analysis - Introduction . . . 60
6.8.1 Analysis Values and Queries . . . 60
6.8.2 Analysis Transition Functions . . . 61
6.9 Analysis Value Transition Rules . . . 62
6.10 Type Rules . . . 65

7 Data Flow Analysis **69**
7.1 Syntax of the Analysis Language . . . 69
7.2 Flow Graph . . . 70
7.3 Analysis Variables of Interest . . . 71
7.4 Partition Sets and Analysis Definition . . . 71
7.4.1 Set Partitions . . . 71
7.4.2 Analysis Values . . . 73
7.4.3 Queries . . . 73
7.4.4 Partition Set Invariants . . . 73
7.4.5 Transition Functions . . . 73
7.4.6 Preconditions . . . 73
7.5 Analysis Values . . . 74

8 Implementation **75**
8.1 Existing Universe Type System with Ownership Transfer . . . 75
8.1.1 Packages . . . 75
8.1.2 Passes . . . 75
8.1.3 Object Structure . . . 77
8.2 General . . . 80
8.3 Local Variable Inference . . . 80
8.3.1 Universe Modifier Translation . . . 80
8.3.2 Java Type Checking and Expression Flattening . . . 80
8.3.3 Analysis Variables of Interest . . . 82
8.3.4 Translation to analysis AST . . . 82
8.3.5 Analysis Solver and Partition Sets . . . 83
8.3.6 Performing the Type Checks . . . 83
8.3.7 Parameter . . . 83
8.3.8 Multiple Universe Types . . . 84
8.3.9 InvTest . . . 85
8.4 Subclass Separation . . . 86
8.5 Arrays . . . 87
8.5.1 Array Type Object . . . 87
8.5.2 Array Access . . . 87
8.5.3 Array Creation Expression . . . 87
8.5.4 Array Initializer . . . 87
8.6 Passing Multiple References on a Free Cluster . . . 88
8.6.1 Type Declaration Control . . . 88
8.6.2 Initial Situation . . . 88
8.6.3 Parameter Order . . . 88
8.7 Changes or New Files . . . 89

9 Conclusion and Future Work **91**
9.1 Examples . . . 91
9.1.1 Enhanced Linked List . . . 91
9.1.2 Hash Table Merging . . . 91
9.1.3 AVL Tree . . . 91
9.2 Conclusion . . . 92
9.3 Future Work . . . 92

A Code Examples **97**
A.1 Enhanced Linked List . . . 97
A.2 Merging of Hashtables . . . 102
A.3 AVL Tree . . . 106

Chapter 1

Introduction

1.1 Background

1.1.1 Ownership Type Systems

Ownership type systems allow the programmer to control aliasing and dependencies by structuring the object store hierarchically into contexts. Each object is owned by at most one other object, called its owner. The ownership relation constitutes a tree order. The set of all objects with the same owner is named context. Objects without an owner are in the root context. The set of objects owned by an object is called its representation.

Most ownership models follow the owner-as-dominator property. This means that all reference chains from an object in the root context to an object x in a different context go through x's owner. Less restrictive is the owner-as-modifier property. An object x can be referenced by any other object, but reference chains that do not pass through x's owner must not be used to modify x. This allows the owner to control modifications of owned objects.

1.1.2 Universe Type System

The Universe type system [4] is a lightweight ownership type system which has been implemented in the JML compiler. It enforces the owner-as-modifier property which means that it differs between read-write and read-only references. Owners can control the modification of owned objects, but not the read access.

Ownership Modifiers. The classification of references is done by using an extended type system. The three keywords `peer`, `rep`, and `any`, called ownership modifiers, can be used in front of the standard Java types. They express the object ownership relative to `this`.

- `peer` denotes a reference to an object in the same context.
- `rep` denotes a reference from an object into its context.
- `any`[1] denotes a reference that might point to objects in any context.

A type in the Universe type system consists of both, the ownership modifier and the Java type. The modifier `peer` is used as default value. `rep` and `peer` types are subtypes of `any` types with the same Java type.

`any` references are read-only. They can only be used for field read and calls to `pure` (side-effect free) methods, not for field update or calls to non-pure methods. Therefore, side-effect free methods are required to be annotated with the keyword `pure`. `peer` and `rep` references are read-write and thus can be used for write access and calls to non-pure methods too.

[1] In earlier descriptions and in currently implemented tools, `any` is called `readonly`

Type Combinator. The type combinator shown in table 1.1 is used to determine the type of transitive access like field access, method parameters, method results, and array element access. For instance if you want to access the field x.f you have to combine the modifier of x with the modifier of f to yield the modifier of x.f from the caller's point of view.

$\triangleright_U$	peer	rep	any
peer	peer	any	any
rep	rep	any	any
any	any	any	any

Table 1.1: Universe type combinator. The left-most cell of the rows means the first argument, the top-most cell of the columns means the second argument.

Runtime System. During runtime each object stores a reference to its owner which is determined by the creation expression. This runtime information is needed for downcasts and to evaluate `instanceof` expressions.

Example. As an example, figure 1.1 and 1.2 illustrate a double-linked list with an iterator. Class *Iterator* shows the power of the owner-as-modifier property. An iterator can have an `any` reference to the nodes of the list and iterate the elements over this read-only reference. Otherwise the *LinkedList* object has full control over the modification on the *Node* objects.

1.1.3 External Uniqueness

Unique variables is a concept for alias management in object-oriented programming. The uniqueness invariant requires that a unique variable, also called unique reference, is either `null` or else its value is the sole reference to an object.

External uniqueness [2] weakens the constraint of unique references. An externally unique reference is the only reference into an aggregate from outside of the aggregate. Internal aliasing to an object from its owner representation is permitted. In combination with ownership types, an externally unique reference is the only reference that passes the context boundary of an object aggregate. If we have a data structure referred by an externally unique reference we can safely transfer it because we can be sure that there is no other external reference into the data structure.

1.1.4 Alias Burying

Alias burying [1] is an approach to maintain the uniqueness invariant. The idea is that aliases of a unique variable do no harm if they are not used anymore to access the referenced object at the moment the unique variable is read again. If a unique reference is read all aliases to the object are marked as unusable and cannot be used to access the object anymore.

1.2 Universe Types with Transfer

To be practical, ownership systems must allow objects to migrate from one owner to another. This ownership transfer is, for instance, desirable in the Abstract Factory pattern or in the case of merging data structures.

Universe types with transfer or UTT for short [11, 12] is an extension of Universe types that supports such ownership transfer. It combines the ideas of external uniqueness and alias burying with the Universe type system. It guarantees statically that a cluster of objects is externally unique when it is transferred and therefore, that ownership transfer is type safe. It provides the same encapsulation as Universe types and requires only negligible annotation overhead.

```
class LinkedList {
        rep Node header;
}

class Node {
        any Object element;
        peer Node next, prev;
}

class Iterator {
        peer LinkedList list;
        any Node current;
}
```

Figure 1.1: Implementation of a linked list with iterator in the Universe type system

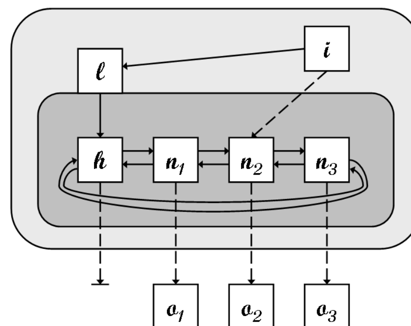

Figure 1.2: Ownership structure of a linked list with iterator in the Universe type system. The nodes h and n_i are owned by the list object l. The iterator has a read-only reference to the nodes. Squares symbolize objects, solid arrows represent read-write references (`peer` or `rep`), dashed arrows denote read-only references (`any`), and the solid rectangles are contexts.

1.2.1 Cluster Transfer

Clusters. To support that only a part of the object's representation is transferred, the ownership context is divided into so-called clusters. These clusters act as the units of an ownership transfer and can therefore only be transferred as a whole. The special `this` cluster may not be transferred. Clusters correspond to externally unique aggregates. Multiple read-write references into a cluster are allowed, as long as it is guaranteed that at the moment of a transfer there is indeed only a single reference pointing into the transferred cluster. Since `any` references are not used for modifications, they do not have to be considered to maintain the uniqueness invariant.

An object is owned by a pair consisting of an owner and a cluster name. Two types `rep` $[f]$ and `rep` $[g]$ are only assignable to each other iff $f = g$.

Cluster Declaration. The ownership modifiers `uniq` and `rep` $[f]$ are used to handle clusters. The definitions of the `rep` and the `peer` modifier have to be adapted to the concept of clusters:

- `uniq` denotes a reference that points into a newly defined transferable cluster. It can only be used for fields.
- `rep`[f] denotes a reference that points into the cluster defined by field f. f must be declared `uniq` in the enclosing class. It may only be used in field declarations and method signatures.
- `rep`
  - for fields and in method signatures: denotes a reference that points into the `this` cluster.
  - for local variables: reference pointing into a "wildcard" cluster. This is explained below.
- `peer` denotes a reference that points into the same cluster.

Local variables that point into the representation of `this` are to be declared solely with a plain `rep` modifier. That can be seen as pointing into an existing cluster which does not have to be specified further. The actual cluster which the variable may point into is statically inferred.

Ownership Transfer. The ownership transfer of a cluster happens in two steps: First, the cluster is released by the current owner, and then it may be captured into an ownership context to complete the ownership transfer. Both operations will take place implicitly, but the additional ownership modifier `free` is needed:

- `free` denotes the only read-write, not unusable reference to a released cluster. It may only be used in method signatures.

The Universe type combinator for the modifier `free` is defined as follows:

$$\text{x} \rhd_U \texttt{free} := \texttt{free}$$
$$\texttt{free} \rhd_U \text{x} := \texttt{any},\ \text{x} \neq \texttt{free}$$

Cluster Alias Controlling. To control external references, analogous to the concept of alias burying, a set of unusable variables that must not be read is used. There are some rules to handle this set:

- Whenever a possible release operation is inferred, all variables that point into the released cluster are marked as unusable.
- Before a non-pure method call on a `peer` receiver, all local variables pointing into a non-free cluster are marked as unusable.
- Reading a `free` variable marks it as unusable.
- Assigning to a variable marks it as not unusable anymore.

1.2.2 Data Flow Analysis

A static, intra-procedural data flow analysis tracks which variables refer into which cluster and which variables are unusable. This is computed for each program step in the method.

Each statement of the language is translated with the so called analysis value transition rules to one or more analysis statements. They represent the modifications this language statement has on the cluster which each variable belongs to.

Flow Graph. The data flow analysis operates on a graph representation of a given analysis statement. This flow graph represents all possible control flow transitions from one elementary analysis statement to another.

Ternary Logic. Due to control structures it is not always possible to allocate one variable to one single cluster as you will see in the following example:

```
uniq T f;
uniq T g;
void m() {
        rep T x;
        if (...)
                x = f; // x is of type rep[f]
        else
                x = g; // x is of type rep[g]
}
```

At the end of the method, x can be either of type `rep` $[f]$ or `rep` $[g]$. Due to these conflict a ternary logic is used for the analysis. This means that the answer to if a variable points into a cluster can be either "YES", "NO" or "DONT_KNOW" which means that the variable may point into this cluster or not.

Analysis Checks. The data flow analysis is employed to check the alias constraints. It performs the following checks:

- No unusable variable must be read.
- No fields must be unusable before a non-pure `peer` method call and upon method termination.

1.2.3 Example

As example figure 1.3 shows the implementation of merging two double-linked lists annotated with the Universe type system. The different steps are illustrated in figures 1.4, 1.5, 1.6, and 1.7. Interesting is the ownership transfer of the list. The first step happens in line 6 in the method *getHeader*. The cluster of field *header* (including all nodes) is released as you can see in figure 1.5. This means that *header* is unusable after this line. Thus, a new *Node* is assigned to *header* in the next line which makes *header* not unusable anymore (see figure 1.6). The second part of the ownership transfer happens in line 20, where the released cluster is implicitly captured (see 1.7) by assigning the externally unique reference to a local variable.

In line 6 the special method *release* is needed to release the *header* node. Otherwise the *header* field would be unusable after the *return* statement, because *header* and *result* point into the same cluster [12, p.128].

For further understanding we list in the code in figure 1.3 additionally the result for the data flow analysis which shows which cluster the `rep` variables *header* and *otherHeader* in the method *merge* as well as the variables *header* and *result* in the method *getHeader* belong to at different points in the methods. For instance in line 21 it means that *header* points into the cluster Cl_header while *otherHeader* points to a separate cluster.

```
1  class LinkedList {
2          uniq Node header = new rep Node();
3
4          free Node getHeader() {
5                  // {CL_header header | unusable}
6                  rep Node result = release(header); // {CL_header | result | unusable header}
7
8                  header = new rep Node();
9                  // {CL_header header | result | unusable}
10
11                 return result;
12         }
13
14         pure free Node release(free Node x) {
15                 return x;
16         }
17
18         void merge(peer LinkedList other) {
19                 // {CL_header header}
20                 rep Node otherHeader = other.getHeader();
21                 // {CL_header header | otherHeader}
22
23                 header.prev.next = otherHeader.next;
24                 // {CL_header header otherHeader}
25                 otherHeader.prev.next = header;
26                 otherHeader.next.prev = header.prev;
27                 header.prev = otherHeader.prev;
28         }
29 }
30
31 class Node {
32         any Object element;
33         peer Node next, prev;
34
35         Node() {
36                 next = this;
37                 prev = this;
38         }
39 }
```

Figure 1.3: Implementation of list merging

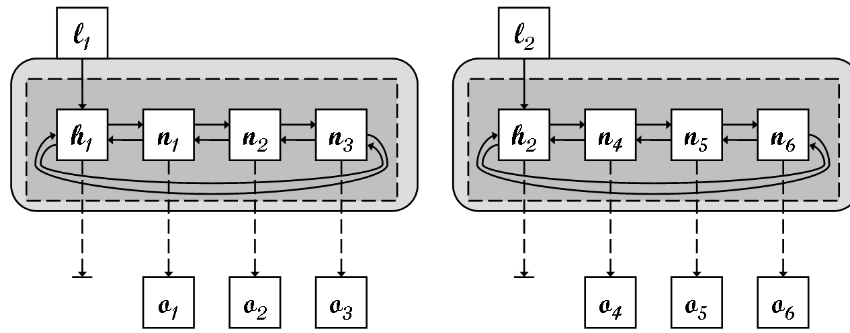

Figure 1.4: Ownership structure of list merging, step 1. This is the initial position. In this and the following figures, squares symbolize objects, solid arrows represent read-write references, dashed arrows denote read-only references, dotted arrows represent unusable references, solid rectangles are contexts, and dashed rectangles denote clusters.

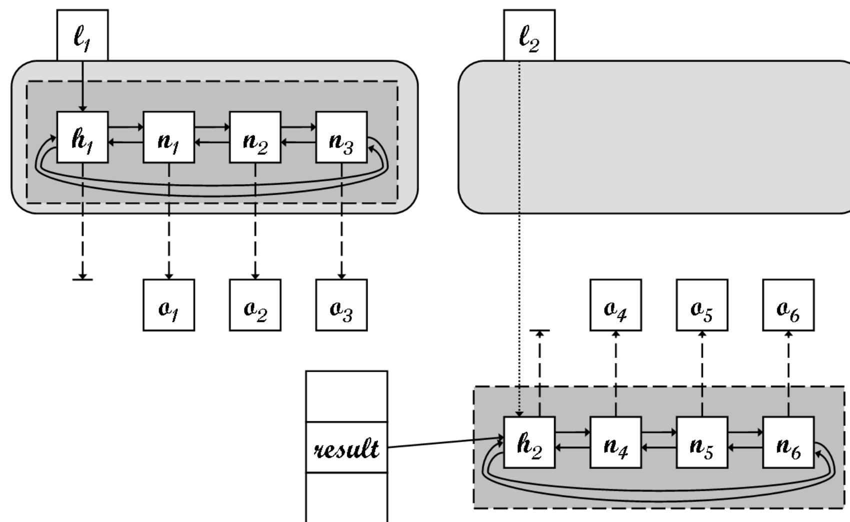

Figure 1.5: Ownership structure of list merging, step 2. The cluster is now released. Thus, the reference from object l_2 to object h_2 become unusable.

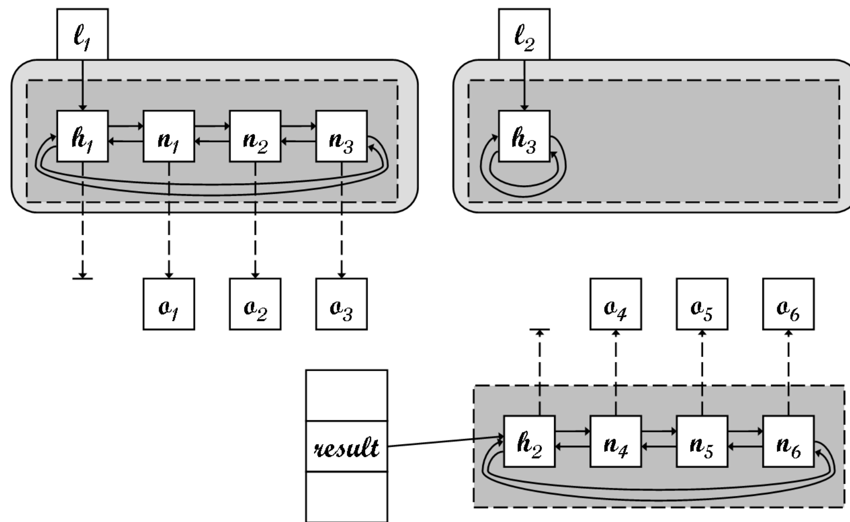

Figure 1.6: Ownership structure of list merging, step 3. We assigned a new node object h_3 to the unusable reference $l2$. Thus it is not unusable anymore.

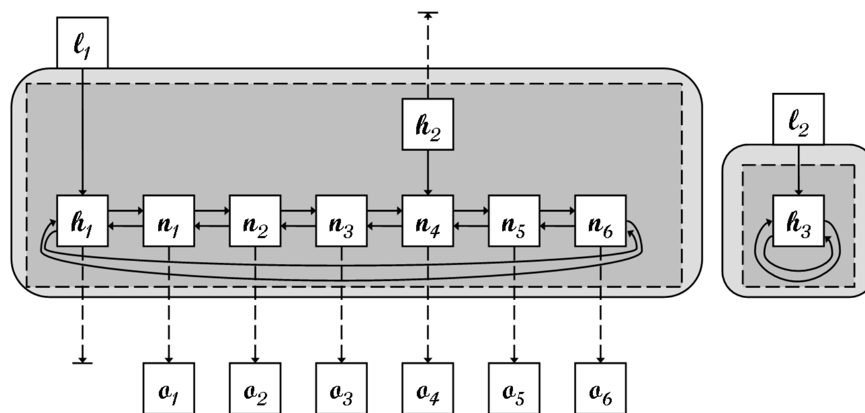

Figure 1.7: Ownership structure of list merging, step 4. Now the free cluster is captured by the list l_1. At the end the object h_2 will be garbage collected because it is not reachable anymore.

1.3 JML and MultiJava

The Universe type system as well as the extensions for ownership transfer have been integrated into the MultiJava compiler and the JML tools. Beside the static type checking it provides runtime and byte code generation support.

1.3.1 JML

JML [10], which stands for "Java Modelling Language", is a behavioral interface specification language designed to specify Java classes and interfaces. It is used to specify the behavior and the syntactic interface of Java code.

JML builds on top of the MultiJava compiler and its utility classes. Thus, most of the uniqueness and ownership transfer extensions are related to the MultiJava code, as JML automatically benefits from them.

1.3.2 MultiJava

MultiJava [3] is a backwards-compatible extension to Java. It retains Java's existing class-based encapsulation properties, type checking, and compilation model. It allows compilation units to be statically type checked modularly and safely, ruling out any link-time or run-time type errors.

1.4 Project Goals

The object of this project is to extend the Universe type system on the presence of ownership transfer. It relies on a recent master's thesis [12] and contains different parts that should be improved.

Local variables are now annotated with an ownership modifier. In case of a `rep` variable the cluster should not be declared because it is inferred statically. In this report we will introduce a possibility to omit any ownership modifier for local variables. Instead we will statically infer the ownership type related with a local variable.

The existing solution uses one cluster (called `this` cluster) for fields annotated with a plain `rep` modifier regardless in which class they are declared. For modular verification it would be nicer if the clusters of a subclass are separated from the clusters of its superclass. Our aim is to introduce an own, non-transferable cluster for each class to hold the subclass separation property.

In [12] the support for arrays is not implemented. Thus, our aim is to find a flexible solution for arrays. Our goal is to handle array element access like field access and thus derive array rules from our field rules. Limitations are given by the restricted information we have about arrays at static time.

In the existing solution it is not possible to specify two formal parameters pointing into the same `free` cluster. For more flexible ownership transfer this would be a nice extension. Therefore, our intent is to support passing of multiple references on a cluster via method invocation.

1.5 Overview

In the next chapters we will spend time on our extensions. In chapter 2 we first care about the type inference for local variables. Chapter 3 will present our solution to separate the cluster of a subclass from the clusters of its superclass. Our result for array handling is given in chapter 4. In the following chapter 5 we illustrate how to pass multiple references on a cluster via method invocation. In chapter 6 we list the formalization of our new type system including all our extensions. The subsequent chapter 7 handles how the data flow analysis works considering all our modifications. Chapter 8 on the one hand shows how the Universe type system with ownership transfer is implemented and on the other hand it documents our changes on this implementation. The final chapter 9 is about conclusion and future work.

All class diagrams in the following chapters are published as UML diagrams. For object diagrams we use a graphic rendition presented in [7].

Chapter 2

Type Inference for Local Variables

2.1 Introduction

In the old solution [12] local variables have to be annotated with one of the three modifiers `peer`, `rep` or `any`, whereas the plain `rep` modifier means that the local variable points into an arbitrary cluster of `this`. The exact cluster is inferred by the analyzer. This solution unburdens the user from declaring cluster details with local variables. Is also gives a higher expressiveness, as demonstrated by this short code fragment:

```
uniq T f;
uniq T g;
void m() {
        rep T x;
        if (...)
                x = f; // x is really of type rep[f]
        else
                x = g; // x is really of type rep[g]
}
```

Static Data Flow Analysis. The inference is effectively done during the static data flow analysis. This data flow analysis is intra-procedural and each run of the analysis is handling one fixed method. Since only clusters of `this` can be transferred, only local variables and fields[1] declared as `rep` can become unusable. Therefore, the analysis needs only to consider `rep` variables and can leave out the ones declared as `peer` or `any`.

If a cluster is transferred in the existing solution, all variables that point into this cluster - excepting the `free` reference - become unusable which means that they must not be read anymore. During the analysis these variables are in an additional cluster marked with *unusable*.

Goal. We will present in this project the solution that all local variables do not need a declared Universe modifier anymore. The Universe type a local variable has will be inferred during the data flow analysis.

[1]We use the term "fields" for instance variables declared within a class and outside a method whereas "local variables" are declared inside a method. "Variables" includes both fields and local variables.

2.2 Solution

2.2.1 Local Variables

In our solution we omit every ownership modifier with local variables at the declaration time as well as in combination with the `new` operation. This means that each local variable can have any type[2] except `uniq` and `free` and will never become unusable. Further local variables have not only one type, but they can change their type. An example of this type switching is shown in the following code snipped[3]:

```
rep T f,g;
peer T h,i;
void m() {
        T x; // { CL_this f g | CL_peer h i | x }
        x = f; // x has at this point type rep[this]: { CL_this f g x | CL_peer h i }
        g = x; // is a correct assignment
        x = h; // x has now type peer: { CL_this f g | CL_peer h i x }
        i = x; // is a correct assignment
}
```

The data flow analysis computes for each local variable to which cluster it belongs at each elementary analysis statement. Thus we have to look up the actual type a local variable has at a certain point in the result of the analysis.

2.2.2 Fields

Moreover we allow input parameters and fields of type `rep` to change their type from `rep` $\langle g \rangle$ to `rep` $\langle f \rangle$. This leads us to a more powerful type system as you can see in the example in section 2.2.11. On the other hand input parameters and fields of type `peer` and `any` cannot change their type. Thus, we distinguish between transferable clusters and non-transferable clusters. Transferable clusters are clusters declared by a field like the cluster declared by `uniq` f. Non-transferable clusters are all other clusters - the `this` clusters as well as the clusters `peer`, `any` and *unusable*. We allow input parameters and fields to change their type from transferable `rep` to an arbitrary `rep`.

Therefore, we know statically which type - `peer`, `rep`, or `any` - a field has, but the current cluster of `rep` fields and input parameters should be inferred by looking up in the result of the analysis instead of taking the declared cluster. Furthermore before calling a method on a `peer` receiver or on `this` we have to restore the object's invariant. This invariant says that fields of different types refer to different clusters and that the type inferred by the data flow analysis is assignable to the declared type.

This short example shows the type switching of fields from one `rep` cluster to another `rep` cluster:

```
uniq T f;
uniq T g;
void m() {
        T x; // { CL_f f | CL_g g | x }
        x = f; // x has at this point type rep[f]: { CL_f f x | CL_g g }
        g = x; // g temporary change the type to rep[f]: { CL_f f x g | CL_g }
        ...
        // restore object's invariant: reassign g
        g = new T(); // { CL_f f x | CL_g | g }
```

[2]We use the term ”type” as short for ownership type. If we mean Java types we explicitly write this term out.

[3]For understandability we list `any` and `peer` fields in the cluster sets in all following examples although they are not really held in the sets. Also we mention the markers for all clusters although in the developed type system we do not use markers for transferable clusters. In addition we leave out not used sets.

```
        this.n();
    }
```

Not allowed is the type switching for fields from `rep` to `peer` as shown in the following code section:

```
    peer T f;
    uniq T g;
    void m() {
        T x; // { CL_peer f | CL_g g | x }
        x = f; // x has at this point type peer: { CL_peer f x | CL_g g }
        g = x; // not allowed since fields of type rep cannot change their type to peer
    }
```

2.2.3 Cluster Transfer

The type switching of local variables means that they cannot become unusable. If we now transfer a cluster containing local variables and fields, the local variables change their type while the fields (as long as it is not a transfer from one `rep` cluster to another `rep` cluster) become unusable. This is demonstrated in the following example:

```
    uniq T f;
    rep[f] T g;
    peer h;
    void m() {
        T x, y;
        // { CL_f f g | CL_peer h | unusable | x | y }
        x = f; // { CL_f f g x | CL_peer h | unusable | y }
        y = f; // { CL_f f g x y | CL_peer h | unusable}
        h = x // merge of the two clusters of h and x
        // the two local variables x and y are transferred to peer
        // while the two fields f and g become unusable
        // { CL_f | CL_peer h x y | unusable f g}
         ...
    }
```

2.2.4 Cluster of New Variables

If we create a new object with the `new` operator we put the corresponding variable in a newly created cluster. This means that it has type `rep` of a transferable cluster (`rep` $\langle Cl_{tr} \rangle$). This is problematic if we want to create a `peer` variable or a variable of a non-transferable `rep` cluster ((`rep` $\langle Cl_{nonTr} \rangle$)). The following example illustrates these circumstances (we here use for the first time the modifier notation rep<Cl> instead of rep[Cl]; the difference between the two is explained in section 6.3):

```
class T {
    peer T p1, p2;
    void m() {
        T x = new T(); // {CL_peer p1 p2 | x}
        x.p1 = p2;     // assignment from peer to rep<CL_tr>
    }
}
```

If we handle x like a `peer` variable the assignment on the second line would be correct. But since x is in a newly created `rep` cluster the assignment would mean an assignment from `peer` to `rep` $\langle Cl_{tr} \rangle$ which is not allowed. The same problem exists with method calls:

```
class T {
        peer T p;

        void m() {
                T x = new T(); // {Cl_peer p | x}
                x.n(p); // expected parameter type is rep<Cl_tr> |> peer = rep<Cl_tr>
                                // actual parameter has type peer
        }

        void n(peer T param) {}
}
```

If x would be a `peer` variable the method call would be correct. But since x is of type `rep` $\langle Cl_{tr} \rangle$ the expected parameter type is `rep` $\langle Cl_{tr} \rangle$ too while the actual parameter type is `peer`.

New non-transferable rep Objects. The same issue occurs if we want to have a new created object of type `rep` of a non-transferable cluster:

```
class T {
        rep T r;
        peer T p;

        void m() {
                T x = new T(); // {Cl_this r | Cl_peer p | x}
                x.p = r;                    // assignment from rep<Cl_this> to rep<Cl_tr>
        }
}
```

and for method invocation:

```
class T {
        rep T r;

        void m() {
                T x = new T(); // {Cl_this r| x}
                x.n(r); // expected parameter type is rep<Cl_tr> |> peer = rep<Cl_tr>
                                // actual parameter has type rep<Cl_this>
        }

        void n(peer T param) {}
}
```

Solution. To solve this conflict we have to allow field writing $y.f = x$ if

$$y = \texttt{rep}\, \langle Cl_{tr} \rangle \wedge\ f = \texttt{peer} \wedge (x = \texttt{peer} \vee x = \texttt{rep}\, \langle Cl_{nonTr} \rangle)$$

even if it is not correct according to the assignable-to relation. This means that we handle y like a variable of type `peer` or `rep` $\langle Cl_{nonTr} \rangle$ instead of the inferred type `rep` $\langle Cl_{tr} \rangle$. This is completely correct because the cluster of y can make this ownership transfer since it is a transferable cluster. The transfer is really done in the *merge* statement of field writing which makes sure that after this point we handle y definitely like a variable of type `peer` or `rep` $\langle Cl_{nonTr} \rangle$.

The same holds for a method invocation $y.mt(z)$ with formal parameter type mt_p. The parameter is correct beside the assignable-to relation if

$$y = \texttt{rep}\,\langle Cl_{tr}\rangle \wedge\ mt_p = \texttt{peer} \wedge (z = \texttt{peer} \vee z = \texttt{rep}\,\langle Cl_{nonTr}\rangle)$$

The correctness is similar to field writing since a method invocation performs a *merge* between the cluster of the expected parameter and the actual parameter cluster.

These conditions flow into the type rules presented in section 6.10.

2.2.5 Cast

In contrast to `new` expression we have to allow and to consider modifiers in connection with a cast between two local variables. For instance a cast between two local variables can be generated during the flattening and is really needed as you can see in the following example

```
any T f;
peer T g;
peer T h;
void m() {
        f = h;

        // the following is the flattened code of g = (peer T) f
        T x,y;
        y = this.f              // y is in the any cluster
        x = (peer T) y;         // x should be explicitly moved to the cluster of peer
        this.g = x;
}
```

If we do not consider the modifier during the cast expression x would be in the `any` cluster and the assignment to g in the next line would not be allowed. Since the original code $g = (peer\ T)\ f$ causes no problems we have to consider modifiers in combination with casts. Anyhow this modifier should only be optional.

if a modifier before a cast is missing in the existing Universe type system the ownership modifier of the static type of the right hand side is taken. This is a good solution for local variables in the new version too. But instead of the static type of the right hand side we have to take the current type calculated by the analyzer.

With the expression $x = (rep[z]\ T)\ y$ it is possible to cast to the cluster the local variable z belongs to at cast time. The precondition is that z is of type `rep` at this point. Similar is the cast to the cluster a field f belongs to with the expression $x = (rep[f]\ T)\ y$. It is important to see that $rep[f]$ does not mean the cluster declared by $uniq\ f$, but the cluster where a field f is at cast time. It is therefore not needed that f is declared `uniq`. f can be any arbitrary field, but must be declared of type `rep`. It is explicitly possible that f is not in the same cluster as it was declared. We will illustrate this slightly confusing situation with an example:

```
any T f;
uniq T g, h;
rep[g] T g2;
rep[h] T h2;

void m() {
        // {CL_any f | CL_g g g2 | CL_h h h2}
        f = g; // {CL_any f | CL_g g g2 | CL_h h h2}
        h2 = g; // {CL_any f | CL_g g g2 h2 | CL_h h}
        g2 = (rep[h2] T) f; // {CL_any f | CL_g g g2 h2 | CL_h h}
        ...
}
```

The cast to the cluster of $h2$ is a cast to the cluster Cl_g which $h2$ temporary belongs to and not to the cluster Cl_h like $h2$ is declared.

2.2.6 Input Parameters

Input parameters are treated like fields in case of declaration and type switching as explained in section 2.2.2. The only difference between the handling of fields and input parameters is that in the object invariant only fields have to be checked.

To treat parameters like fields we have to insert the keyword `this` before each occurrence of input parameters such that writing and reading from and to input parameters are handled like field reading and writing:

```
void m(peer T p1, rep T p2) {
        T x, y;
        ...
        // before x = p1;
        x = this.p1;
        // before p2 = y;
        this.p2 = y;
}
```

2.2.7 Return Values

For declaration we handle return values like fields. In the old version a `return` statement `return` x was handled like a method call `this`.m(x) where the formal parameter of m has the same type as the declared return type. Since the treatment of parameters is similar to the treatment of return values we can handle the `return` statement like a method call further on.

2.2.8 Static Methods

In the original Universe type system for static methods no `rep` types are allowed in the signature as well as for local variables because there is no receiver object [5]. We extend this restriction and allow `free` parameters (including multiple references to a `free` cluster as introduced later in chapter 5) and return values. Since local variables have no declared type there are no restrictions for local variables anymore.

2.2.9 Purity Checking

We would also improve purity checking. In the old version in a `pure` method only assignment to a local variable, field reading and call to `pure` methods is allowed. We will improve this to allow field writing and call to non-pure methods in some cases.

We can allow field writing $x.f = y$ and invocation of a non-pure method $y = x.mt(z)$ if no fields of the current object are modified. We prevent this by the precondition that x has to be `rep` $\langle Cl_{tr} \rangle$. Thus, no call back to the current object and modification of it is possible since the reference to the current object is `any`. In addition there should be no field in the same cluster as x. This means that x has no read-write reference to a field of `this`.

In addition for a method invocation we have to be sure that x is still of type `rep` $\langle Cl_{tr} \rangle$ after call because as mentioned in section 2.2.4 x can change the type to `peer` or `rep` $\langle Cl_{nonTr} \rangle$. This can be shown by the condition that z should be assignable to x combined with the formal parameter (for further understanding we refer to section 2.2.4). Furthermore we have to ensure that no field of the current object can be accessed through a read-write reference by the actual parameter z. All these restrictions lead us to the following precondition for field writing $x.f = y$ (L^x is the cluster which x belongs to; f' means a field of the current object):

$$x = \texttt{rep}\,\langle Cl_{tr}\rangle \;\wedge$$
$$\nexists f' : f' \in L^x$$

and for call of a non-pure method $y = x.mt(z)$:

$$x = \texttt{rep}\,\langle Cl_{tr}\rangle \;\wedge$$
$$\nexists f' : f' \in L^x \;\wedge$$
$$z \leq_A x \rhd_U mt_p \;\wedge$$
$$mt_p = \texttt{any} \vee (z = \texttt{rep}\,\langle Cl_{tr}\rangle \wedge \nexists f' : f' \in L^z)$$

Thus, `this` and all its fields are only accessible over an `any` reference from x and cannot be modified. We illustrate it by the following example:

```
1   class T{
2           peer T g, f;
3           rep T r;
4
5           void m() {
6                   g = new T();
7           }
8
9           pure void n() {
10                  T x,y;
11                  x = new T();
12                  y = x;
13                  // {Cl_peer g f | x y | Cl_this r}
14                  x.m(); // {Cl_peer g f | x y | Cl_this r}
15                  x.g = f; // {Cl_peer g f | x y | Cl_this r}
16          }
17
18          pure T() {}
19  }
```

The method call in line 14 and the field writing in the subsequent line can be without any problems since they cannot modify the fields of the current object.

In figure 2.1 you can find another example. Inside of the `pure` function *find* at line 10 and 16 we call the non-pure method *getNext*() on the parameter *it*. *it* is of type `rep` and since there is no field of the current object in the same cluster, we allow this method invocation. The whole code example is listed in A.1.

Purity Definition We use in our type system a weaken purity. We define purity in that way that no modifications on objects which are not transitively owned by the current object is allowed. Thus, modification of all objects inside the representation context of the current object is allowed.

2.2.10 free Variables

Parameters and return values can be of type `free`. This needs some special kind of handling. When we have a method call $y.m(z)$ we normally merge the cluster the actual parameter z has with the cluster of the expected type computed by the type combinator between y and mt_p, the formal parameter type: $merge(z, y \rhd_U mt_p)$. In other words we merge the cluster where the parameter is now with the cluster where the parameter would be at the beginning of the method call. As long as the formal parameter type is of a type with a defined cluster this merge leads to no problems.

```
1  class LinkedList {
2          uniq Node header;
3
4          pure free LinkedList findAll(any Element obj){
5                  Iterator it = new Iterator(this);
6                  return find(it, obj);
7          }
8
9          pure free LinkedList find(free Iterator it, any Element obj) {
10                 Node curNode = it.getNext();
11                 LinkedList found = new LinkedList();
12                 while (curNode != header) {
13                         if (curNode.element.equals(obj)) {
14                                 found.addLast(curNode.element);
15                         }
16                         curNode = it.getNext();
17                 }
18                 return found;
19         }
20 }
21
22 class Iterator {
23         peer LinkedList list;
24         any Node current;
25
26         any Node getNext() {
27                 current = current.next;
28                 return current;
29         }
30
31         pure Iterator(peer LinkedList l) {
32                 list = l;
33                 current = l.header;
34         }
35 }
36
37 class Element {
38         int value;
39
40         pure boolean equals (any Element other) {
41                 if (value == other.value)
42                         return true;
43                 else
44                         return false;
45         }
46 }
```

Figure 2.1: Implementation of iterator function which utilizes the enhanced purity.

If the formal parameter type is `free` this means that at the beginning of the method the parameter is in a new cluster, it would then be transferred to an unknown cluster. Thus, since we do not know what kind of transfer is done inside of m we have - by checking the method modular - no knowledge about the type of field f and all other variables in the same cluster after the method call. This conflict is illustrated by the following example:

```
class T {
        peer T g;
        uniq T f;

        void n() {
                g.m1(f); // f is peer at the end of the call
                 ...
                g.m2(f); // f is rep at the end of the call
        }

        void m1(free T p) {
                g = p;  // p becomes peer
        }
        void m2(free T p) {
                f = p;  // p becomes rep
        }
}
```

Both methods $m1$ and $m2$ have the same signature but in $m1$ parameter p has type `peer` at the end of the method while in $m2$ the parameter p has type `rep`. This shows that if we call a method with a parameter of type `free` we have no knowledge about the cluster of this parameter at the end of the method call.

Solution. To prevent access to this parameter and all variables in the same cluster, we merge the cluster of f with the `any` cluster. This means that all fields in the cluster of f become unusable and all local variables in this cluster change their type to `any`.

Thus, in the following example we do not allow the assignment inside of n since f becomes unusable after the method call:

```
        peer T g;
        uniq T f;
        rep[f]  T h;

        void n() {
                g.m(f);
                h = f; // type error since f is unusable
        }

        void m(free p) {}
```

Free Return Values. We have a similar situation if we are inside a method and return a `free` variable:

```
        free n() {
                return x;
        }
```

We handle a `return` statement like a method call. Thus, we have to merge the cluster of x with the cluster of the expected return type. By the same token as for `free` parameters we do a merge of x with `any` if the expected return type is `free`.

2.2.11 Releasing Problem

In the old version there exists a problem that sometimes a special release function was needed to release a reference [12, p. 128]. This fact is shown in the following example deduced from the list merging example:

```
uniq Node header;

pure free Node release(free Node x) {
        return x;
}

free Node getHeader() {
        // {CL_header header | unusable}
        rep Node result = release(header);
        // {CL_header | result | unusable header}
        header = new rep Node(null, null, null);
        // {CL_header header | result | unusable}
        return result;
}
```

The reason for the need of the additional method *release* was that the type system enforced a one-to-one correspondence between static clusters (a set of fields) and dynamic clusters (a set of objects). This means that if we would use a direct assignment statement:

```
uniq Node header;

free Node getHeader() {
        // {CL_header header | unusable}
        rep Node result = header;
        // {CL_header header result | unusable}
        header = new rep Node(null, null, null);
        // {CL_header header result | unusable}
        return result;
}
```

header would be in the same cluster as *result* before the `return` statement and would therefore become unusable since the result type should be `free`. Thus, we needed the *release* method mentioned above.

With our solution and the local variable inference we have solved this minor hitch. Since the newly created *header* object is in a new cluster instead of the declared cluster, *result* and *header* are not in the same cluster at the end of the method:

```
uniq Node header;

free Node getHeader() {
        // {CL_header header | unusable}
        Node result = header;
        // {CL_header header result | unusable}
        header = new Node(null, null, null);
        // {CL_header result | header | unusable}
        return result;
```

```
}
```

This example shows that with our solution we have a more flexible handling of clusters which allows easier handling of some examples as shown above. The disadvantage of this flexible handling is, that each newly declared cluster needs at least one field that is declared as pointing into it. This condition is satisfied by our type system since the only way to declare a new cluster by using the keyword `uniq`.

2.2.12 Summary

Subsumed we can say that:

- Local variables have no fixed type and can therefore change their type freely after each statement.
- Local variables can never become unusable.
- Input parameters and fields of type `peer` and `any` cannot change their type.
- Input parameters and fields of type `rep` $\langle Cl_{tr} \rangle$ can change the type on `rep`[_] or on *unusable*.
- Before a method call on a `peer` receiver or on `this`, fields of different statically types should be in different clusters.
- For all local variables we need to look up the current types in the result of the analysis.
- For `peer` and `any` fields and input parameters we can take the statically declared types since they cannot change their type. Thus, we can leave out these fields and input parameters for the data flow analysis since we know which cluster they belong to.
- For fields and input parameters of type `rep` we need to look up the current cluster in the result of the analysis.
- We can allow field writing and call to non-pure methods inside of `pure` methods if the receiver object has type `rep` and there is no field in the same cluster as the receiver object.

Chapter 3

Subclass Separation

3.1 Introduction

In the old version a class and its subclass share the clusters. In the following code example the fields f and g are both in the same `this` cluster:

```
class A {
        rep A f;
}

class B extends A {
        rep A g;
}
```

It is correct to handle the `this` cluster in such a way, but it would be better if we separated the clusters of a class from the clusters of its subclass. Motivation for this subclass separation is for instance the static verification process in Spec# [9]. Each object in Spec# has to fulfill the invariants of all its superclasses. For example an object x of type B which is a subtype of A should hold both invariants $inv_A \wedge inv_B$. During the verification process it is sometimes desirable that we check only the invariant inv_B without touching the fields that x inherits from A. Thus, Spec# needs the separation of subclasses.

Goal. In our solution we want to separate the cluster of one class from the clusters of its subclass.

3.2 Solution

3.2.1 Subclass Separation

To allow modular checking a method should be sure that no fields of subclasses can belong to the clusters it analyzes. In other words fields of a subclass may not be declared of clusters used in the analysis of the superclass.

Since we include only `rep` fields in our analysis, the `peer` and `any` cluster can only be used by local variables and thus only by the class where the method is declared. The `rep` clusters declared by a `uniq` field can only be accessed by the `rep` variables of the class where the field - and thus the cluster - is declared. Thus, the only two clusters that are shared by a class and its subclasses are the `this` cluster and the *unusable* cluster. If we have subclass separation this means that the fields of a subclass cannot be in the cluster used by the superclass. Therefore no field of the subclass can become *unusable*. This means that we only need to separate the `this` cluster.

To separate the clusters our solution does not work with one `this` cluster but with one single `this` cluster per class. All `this` clusters are marked with a corresponding marker and thus are non-transferable. This avoids the merging of two `this` clusters.

3.2.2 Notation

The notation will stay as in the old version. For fields, parameters and result types `rep` means that the variable belongs to the `this` cluster of the class where the variable is declared. Due to the absence of the possibility to declare explicitly to which `this` cluster a variable belongs there is no additional check needed to ensure that variables use only the `this` cluster of the class they were declared in.

3.2.3 Cluster for the Analysis

In the analysis we have - beside the `peer` cluster, the `any` cluster, the *unusable* cluster and the clusters declared by fields of the current class and all its superclasses - one `this` cluster for the current class and one `this` cluster for each superclass of this class. The `this` cluster of the superclasses as well as the clusters declared by fields of the superclasses are only needed in the analysis and cannot be used for variable declaration in the current class.

3.2.4 Changes in the Semantic

Due to these changes to achieve the subclass separation some examples that work smoothly in the old version are not correct anymore. The following example illustrates this problem:

```
1  class A {
2          rep A g;
3          void n(rep A param) {
4          }
5  }
6
7  class B extends A {
8          rep A f;
9          void m() {
10                 n(f);
11         }
12 }
```

The method invocation at line 10 requires the merging of the clusters of the actual parameter *f* and the formal parameter *param*. Without subclass separation *f* and *param* are in the same `this` cluster while they are in the presented solution in two different `this` clusters. Since all `this` clusters are non-transferable, the method invocation enforces a merging of two non-transferable clusters which is not allowed. In other words according to the new solution this example is not well-typed which it was in the old solution.

Another example which was correct in the existing system and is not well-type anymore by using subclass separation is the following:

```
class A{
        rep Node head1;
}

class B extends A{
        rep Node head2;

        void SubclassSeparationViolation(){
```

```
            // {CL_A head1 | CL_B head2}
            if (head1 == null || head2 == null)
                return;
            Node cur = head1;
            // {CL_A head1 cur | CL_B head2}
            while (cur.next != null)
                cur = cur.next;
            // {CL_A head1 cur | CL_B head2}
            Node end1 = cur;
            // {CL_A head1 cur end1 | CL_B head2}
            cur = head2;
            // {CL_A head1 end1 | CL_B head2 cur}
            while (cur.next != null)
                cur = cur.next;
            // {CL_A head1 end1 | CL_B head2 cur}
            end1.next = cur; // violation: cur is not assignable to end1.next
        }
}

class Node {
        any Object element;
        peer Node next;
}
```

Class A and class B both have a list in their `this` cluster referred to by the fields $head1$ and $head2$. At the last line of the method $SubclassSeparationViolation$ we try to create a reference from the end of the first list to the end of the second list. This fails since one list is in the cluster Cl_A and the other in the cluster Cl_B and both clusters are non-transferable

Overloading and Overriding. A third example shows the changes in connection with method overriding and overloading:

```
class A {
        void n(rep A param) {
        }
}

class B extends A {
        void n(rep A param) {
        }
}
```

Class B tries to override a method of class A with a plain `rep` parameter. Since both methods have semantically not the same signature - in class A the parameter has type `rep` $\langle Cl_A \rangle$ and in class B `rep` $\langle Cl_B \rangle$ - this is not handled like a case of overriding but like a case of overloading. Due to the Universe type system rules [5] overloading of methods is forbidden if the signatures only differ in their ownership modifier. Thus, our example is not allowed. This means that the rule about overloading prevents us from critical situations in connection with the subclass separation since `rep` in one class is handled as a different type than `rep` in another class.

3.2.5 Summary

We can subsume the subclass separation as follows:

- We create one `this` cluster for each class.

- All `this` clusters are non-transferable.
- A variable of type `rep` points always into the `this` cluster of the class where it was declared.
- Two `rep` types of two different classes A and B are handled like different types `rep` $\langle Cl_A \rangle$ and `rep` $\langle Cl_B \rangle$.

Chapter 4

Arrays

4.1 Introduction

In the existing system there was no solution to handle arrays with ownership transfer. Thus we give here a little introduction to the handling of arrays in the normal universe type system [5].

Arrays of reference types need two ownership modifiers: one for the array object and one for the type of the reference they store. If only one modifier is present it is taken for the element type and the array object type is by default `peer`. The modifier of the elements has to be `peer` or `any`. `rep` is not allowed. Arrays of primitive types need only the modifier for the array object.

Access to arrays is basically interpreted like field access, this means that the type combinator is used to determine the type of the access expression. For write access we need runtime check.

We have covariant array subtyping. The type hierarchy can be seen in figure 4.1.

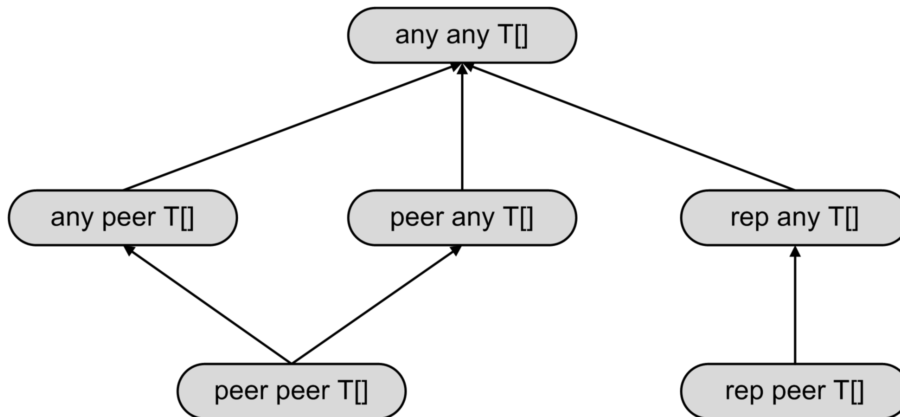

Figure 4.1: Type hierarchy for arrays in the Universe type system.

Goal. Our aim is to find a solution which allows a flexible handling of arrays. It is limited by the information we have about arrays at static time.

4.2 Solution

4.2.1 Idea

For arrays we have to look at the array object as well as the array elements. There is a strong parallelism between field access and array access which we will demonstrate by the following example:

```
1  class T {
2          peer T a0, a1, a2;
3  }
4
5  class A {
6          uniq T g;
7          rep[g] peer A[] array;
8
9          void m() {
10                 T x = new A();
11
12                 // array object access
13                 array = x; // similar to g = x
14                 x = array; // similar to x = g
15
16                 // array element access
17                 array [0] = x; // similar to g.a0 = x
18                 x = array [1]; // similar to x = g.a1
19         }
20 }
```

The variable *array* can be compared with the field g. Both are of type `rep`[g] and syntactically there is no difference between $array = x$ and $g = x$ as you can see in line 13 and 14. For array element access in lines 17 and 18 you find a similar situation. The expression $array[0] = x$ is comparable with the expression $g.a0 = x$. In both expressions we access through a `rep` object an element that is `peer` to the prefix. Clearly the syntax is diverse in both cases. Beside the different syntax the only difference between array and field access is that in the case of fields we know the number of "elements" at static time while we do not know this about the array. By the same token we know statically always which field we access, but we do not always have the cognition which array element is accessed. Due to the parallelism between fields and arrays we try to derivate the solution for arrays from our handling of fields.

We will call arrays declared directly in a class "field arrays", array declared in a method "local variable arrays" and arrays used as parameter "parameter arrays".

4.2.2 Array Declaration

The use of the ownership types at declaration is similar to the other variables. A field array and a parameter array can be declared with two types, one for the array object and one for the elements as explained in section 4.1.

Otherwise the type of a local array object is inferred and thus needs no declaration. The modifier for the elements - which can only be `peer` and `any`, but never `rep` - can be declared. If it is not declared, `peer` is taken as default value. Notice that the modifier for local variable array elements is the only case where it is allowed to declare local variables with an ownership modifier.

4.2.3 Assignable-to Relation

The assignable-to relation for array types is listed in figure 4.2. Since we can transfer the cluster we allow assignment from an array object of type `rep` $\langle Cl_{tr} \rangle$ to an array object of type `rep` $\langle Cl_{nonTr} \rangle$ as long as the elements are assignable too.

4.2.4 Array Object

As we have listed above, array objects can be handled like normal fields, parameters or local variables. The only difference is that we have to consider the assignable-to relation for arrays.

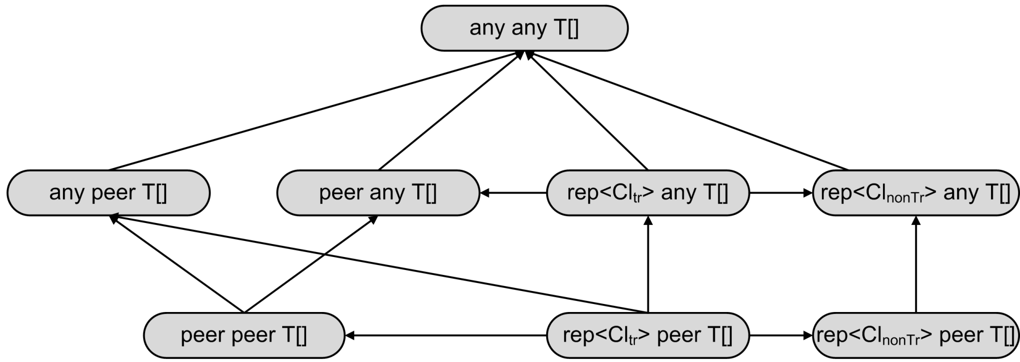

Figure 4.2: Assignable-to relation for arrays. Cl_{tr} means a transferable cluster whereas Cl_{nonTr} is a non-transferable cluster.

Analysis Variable. We create for each array object of a local variable array as well as for each field array and parameter array of type `rep` an analysis variable to observe the cluster the reference belongs to.

4.2.5 Array Elements

Statically we have not always information about the length of the array as well as which array element is accessed. If we want to handle each array element separately, different problems appear as we explain by the example below:

```
uniq f, g;
rep[f] peer T[] ar;

void m(int i) {
        //{CL_f f ar ar[0] ... ar[n] | CL_g g}
        ar[i] = g;
        //{CL_f f ar ar[0] ... ar[i] ar[i+2] ... ar[n] | CL_g g ar[i+1]}
        ...
        ar[i] = f;
        //{CL_f f ar ar[0] ... ar[i] ar[i+1] ar[i+2] ... ar[n] | CL_g g}
}
```

First we do not know the length of the array and thus cannot determine how many array elements should be handled in the analysis. The method m get an integer i as parameter. Since we statically do not know the value of i we cannot decide which array element is moved to cluster Cl_g. Thus, when we want to restore the object's invariant we do not really know which array element should be moved from cluster Cl_g back to cluster Cl_f. All this missing information is marked in the example with the variables i and n. As long as we do not know the values of these two variables it is not possible to handle each element of an array separately in the analysis. This means that we have to handle all elements like one single object. Consequently all elements of an array always have to be in the same cluster.

This means that we have to care about one element object for each array which we will call below the array element object. Thus, $array[0]$ and $array[1]$ will have the same effect.

Types. The Java type of the array element object is always the type of the array elements. The ownership type of the array element object is computed by the combination of the array object type and the array element type. This means that an array declared as `peer peer` has an array element object of type `peer` and an array declared as `rep peer` has an array element object of type `rep`.

Analysis Variable. In the same way as the fields of a field are not integrated in the analysis it is not needed to observe the type of array elements separately. Because we know statically the type of the elements relative to the array object we can get the cluster which the elements belongs to without problems. If the elements are declared `any` they are in the cluster `any`. Otherwise if they are declared as `peer` they are always in the same cluster as the array object whose cluster we can determine by the corresponding analysis variable. Because elements cannot be declared as `rep` there are no more cases. This means that we need no analysis variable to observe the cluster the array elements belong to.

The example introduced above will now look as follows:

```
uniq f, g;
rep[f] peer T[] array;

void m(int i) {
        // {CL_f f array | CL_g g}
        array[i] = g;
        // {CL_f f | CL_g g array}
        ...
        array = new T[array.length];
        // {CL_f f array | CL_g g}
}
```

At the assignment $array[i] = g$ we move the array object - and thus all array elements too - to cluster Cl_g. To restore the object's invariant we have to move the whole array back e.g. by a new expression.

4.2.6 Flattening

During flattening an expression is split into some basic statements by using temporary variables. The flattening of array element access works like the flattening of field access. In the following example:

```
peer T p1, p2;
peer peer T a, b;
int i, j;

void m() {
        T x;
        x = p1.a[1].p2;
        a[1] = b[3];
        x = a[i + j];
}
```

$x = p1.a[1].p2$ is flattened to:

```
tmp0 = this.p1;
tmp1 = tmp0.a;
tmp2 = tmp1[1];
x = tmp2.p2;
```

$a[1] = b[3]$ is flattened to:

```
tmp3 = this.a;
tmp4 = this.b;
tmp5 = tmp4[3];
tmp3[1] = tmp5;
```

and $x = a[i + j]$ is flattened to:

---

tmp6 = **this**.a;
tmp7 = **this**.i;
tmp8 = **this**.j;
tmp9 = tmp7 + tmp8;
x = tmp6[tmp9];

In general the flattening which was already supported by the old version is done as follows:

flatten(prefix[accessor]):
(s_prefix, v_prefix) = flatten(prefix)
(s_accessor, v_accessor) = flatten(accessor)
return ([s_prefix; s_accessor;], v_prefix[v_accessor])

The first result value of the method *flatten* means the flattening - the basic statements - and the second result value means the flattened expression. This is the local variable that represents the result after executing the flattening. This means that we only have to look how to handle array reading of the form:

$$x = y[z];$$

and array writing of the form:

$$y[z] = x;$$

4.2.7 Array Reading

For each basic language statement we declare so called transition rules. They point the change that happens on the state of the clusters when we execute the corresponding statement while L means the cluster state before and L' the cluster state after the execution. The transition rules for all basic statements as well as the used formalization can be found in section 6.9.

Array reading of the form $x = y[z]$ can be seen like field reading $x = y.f$ where f has the same type as the array elements. Thus, we move x to the cluster of the array elements. This cluster is either the cluster of y - if the element type is `peer` - or the `any` cluster if the elements are declared as `any`. This leads us to this simple transition rule:

$$\frac{\begin{array}{c} if(m_{yElem} = peer)\ then\ move(L, x, y) \\ else\ move(L, x, Cl_{any}) \end{array}}{\Gamma; L \vdash_L x = y[z]; L'} \quad \text{[L-ARR-RD]}$$

The so called type rules check whether a language statement is well typed according to our type system or not. Again L represents the cluster state before the execution of the statement. The whole formalization and all type rules are shown in section 6.10. In the type rules there is no check needed. Array reading should always be allowed.

4.2.8 Array Writing

Array writing of the form $y[z] = x$ can be compared with field writing $y.f = x$ where f has the same type as the array elements. If we adapt the field writing transition rule we achieve the following rule for array writing:

$$\frac{if(m_{yElem} \neq \texttt{any})\ merge(L, x, y)\ else\ L}{\Gamma; L \vdash_L y[z] = x; L'} \quad \text{[L-ARR-WR]}$$

We have to keep the condition that `peer` elements stay in the same cluster as the array object. Since we never move the array object but only merge it we hold this condition. The array field writing type rule would look as follows:

$$\frac{\begin{array}{c} y \neq \texttt{any} \\ (x \leq_A y \rhd_U m_{yElem}) \vee \\ (y = \texttt{rep}\,\langle Cl_{tr}\rangle \wedge\ m_{yElem} = \texttt{peer} \wedge\ (x = \texttt{peer} \vee x = \texttt{rep}\,\langle Cl_{nonTr}\rangle)) \end{array}}{\Gamma; L \vdash y[z] = x} \quad \text{[T-ARR-WR]}$$

4.2.9 Unusable Arrays

Although we do not observe which cluster the array elements belong to, access to unusable array elements is prevented by the type rules. If the array elements are of type `any` they can never become unusable. Otherwise if we have `peer` elements they are in the same cluster as the array object. This means that if the array elements are unusable, the array object is in the case of a field or parameter array unusable too or it is `any` in the case of a local variable array. If the array object is unusable, the access to its elements is forbidden by the type rules. On the other hand if the local array object is of type `any`, array element writing is not allowed too. But according to the type rules we allow array reading if the local array object is `any` and thus the elements could be unusable:

```
1  class T {
2          void m() {
3                  T y;
4                  peer T[] x;       // {CL_any | x | y}
5                  x = new T[3];// {CL_any | x | y}
6                  n(x); // {CL_any x | y}
7                  y = x[1];
8          }
9
10         void n(free T[] param) {}
11 }
```

In line 7 we allow array reading, but if we would have observed a single analysis variable for the array elements this would be unusable.

It corresponds to the following field reading situation:

```
1  class T {
2          peer T p;
3
4          void m() {
5                  T x,y; // {CL_peer p | CL_any | x | y}
6                  x = new T();// {CL_peer p | CL_any | x | y}
7                  n(x); // {CL_peer p | CL_any x | y}
8                  y = x.p;
9          }
10
11         void n(free peer T param) {}
12 }
```

The field reading in line 89 is allowed although if we had an individual analysis variable for x.p it would have become unusable in line 7. Since in the field reading as well as in the array reading situation the access is only possible through an `any` reference, only reading is allowed. Reading of an unusable reference should be no problem according to the owner-as-modifier property.

A second critical situation could occur if the array elements become unusable. Since we have no control about which element is accessed we do not have the possibility - expecting with complex methods - to observe whether we assigned a new value to each element of the array. This would be necessary to make the array element object not unusable anymore. Fortunately we do not have

to care about this. As mentioned above only `peer` elements can become unusable and when they are unusable their array object is `any` or unusable too. If we want to make the array elements not unusable anymore while their array object stays `any` or unusable we extract no additional functionality since the access to the elements stays restricted by the array object. Therefore, we have no reason to try making array elements not unusable anymore without touching the array object itself.

Otherwise if the array object changes its cluster - because a new value is assigned - the newly assigned array has its own elements which cannot be unusable as long as the array object is neither `any` nor unusable. This means that we achieve to receive array elements that are not unusable anymore by assigning another array to their array object. We demonstrate this constitution by a little example:

```
1   class T {
2           rep peer T[] a;
3           void m() {
4                   T y;
5                   // {CL_T a | CL_any | y | unusable}
6                   n(x); // {CL_T | CL_any | y | unusable a}
7                   a = new T[3]; // {CL_T | CL_any | a | y | unusable}
8           }
9
10          void n(free T[] param) {}
11  }
```

After line 6 the array object a and its array elements are unusable. With the new array expression in line 7 a becomes not unusable anymore. Since the new array has new elements the elements are not unusable too.

4.2.10 Array Creation Expression

An array can be created by an array creation expression like $T[]\ x = new\ T[i]$. Like for new object creation we do not expect modifiers in combination with the new expression. If there are modifiers, they are omitted and a warning is raised.

Flattening. In an array creation expression, first the dimension expressions are evaluated, left-to-right [8, p. 432]. After that, the space for the new array is allocated. This means that for the flattening we first have to handle the dimension expression and then the array creation itself.

The basic form of an array creation expression is:

$$new\ T[y_1]...[y_i];$$

To achieve this form we first have to flatten each dimension expression and create a new array creation expression with the flattened expression:

$$\text{x = new T[g.i][getInt(i)]}$$

is flattened to:

$$\begin{array}{c}\text{tmp0 = this.g;}\\ \text{tmp1 = tmp0.i;}\\ \text{tmp2 = getInt(i);}\\ \text{x = new T[tmp1][tmp2];}\end{array}$$

Then we create a new assignment from the new array creation expression to a new temporary variable:

$$\text{x = new T[tmp1][tmp2];}$$

is flattened father to:

tmp3 = new T[tmp1][tmp2];
x = tmp3;

More formally the flattening rules looks as follows:

flatten(new T[expr_1] ... [expr_k]):
(s_i, v_i) = flatten(expr_i)
return ([s_i; ...; s_k; tmp = new T[v_1] ... [v_k]], tmp)

In the example above the flattening is:

tmp0 = this.g;
tmp1 = tmp0.i;
tmp2 = getInt(i);
tmp3 = new T[tmp1][tmp2];
while the flattened expression is the temporary local variable *tmp*3.

Transition Rules. The transition rule for array creation expression is derived from the new object creation rule:

$$\frac{L' = new(L, x)}{\Gamma; L \vdash_L x = new\ T[x_1]...[x_i]; L'} \quad \text{[L-NEW-ARR]}$$

Notice that we do not have to care about the dimension expression in this basic form because they are handled in their statements of the flattening.

Type Rules. The type rule for array creation is simple since we do not have to do any type check:

$$\frac{}{\Gamma; L \vdash x = new\ T[x_1]...[x_i];} \quad \text{[T-NEW-ARR]}$$

Like for the transition rules we can ignore the dimension expressions at this point since they are type checked in the statements of the flattening.

4.2.11 Array Initializer

The second possibility to create an array is the array initializer of the form $T[]\ x = \{y, z\}$ or for the two dimensional array $T[][]\ u = \{\{x, y, z\}, \{a, b, c\}\}$. It is a combination of array creation and array writing. Thus we can handle the following array initializer:

```
T[] x = {a, b, c};
```

like:

```
T[] x = new T[3];
x[0] = a;
x[1] = b;
x[2] = c;
```

Flattening. The expressions in an array initializer are executed from left to right in the textual order they occur in the source code [8, p.291]. We can thus flatten an array initializer expression into an array creation expression and some array writing expressions:

$$\text{flatten}(\{x_0, ..., x_k\}):$$
$$(s_i, v_i) = \text{flatten}(x_i)$$
$$\text{return } ([\text{tmp} = \text{new } T[k]; s_0; ...; s_k; \text{tmp}[0] = v_0; ...; \text{tmp}[k] = v_k]; \text{tmp})$$

The rule is shown for a one-dimensional array. For multi-dimensional arrays the rule will recursively go on since $flatten(x[i])$ will flatten the next inner array initializer.

By splitting one array initializer expression into one array creation expression and some array writing expression we achieve that no array initializer expression occurs in the flattened tree. Thus, no transition rules and type checking rules for array initializers are needed.

4.2.12 Array Transfer

Since we handle arrays similar to fields it causes no problems to transfer arrays from one cluster to another. As an example, in figure 4.3 the merging of two hash tables - represented by arrays - is shown. In line 12 we release the array table of *otherTable* by an explicit release function. This released array is passed to method *merge* which expects a `free` array as parameter. The whole code of this example is listed in section A.2.

4.2.13 Summary

Although we do not have all information about an array at static time we reach a good solution for the handling of arrays without radical restrictions. Subsumed we can say that:

- The array object can be handled like a normal variable, only the assignable-to relation is different.
- The modifier of the array elements should always be known at declaration time.
- Array reading and writing are similar to field access.
- All elements of one array must always be in the same cluster.

```
1  class HashTable {
2          uniq peer LinkedList[] hashTable;
3          static int size = 5;
4
5          void merge(free peer LinkedList[] otherTable) {
6                  for (int i = 0; i < hashTable.length; i++) {
7                          hashTable[i].concatenate(otherTable[i]);
8                  }
9          }
10
11         void merge(peer HashTable otherTable) {
12                 merge(otherTable.getHashTable());
13         }
14
15         private static pure int getHashValue(readonly Element e) {
16                 return ((e.value \% size));
17         }
18
19         public free peer LinkedList[] getHashTable() {
20                 peer LinkedList[] res = hashTable;
21                 hashTable = new LinkedList[size];
22                 for (int i = 0; i < hashTable.length; i++) {
23                         hashTable[i] = new LinkedList();
24                 }
25                 return res;
26         }
27
28         public static void main( String [ ] args ) {
29                 HashTable t1 = new HashTable();
30                 HashTable t2 = new HashTable();
31                 t1.merge(t2);
32         }
33 }
```

Figure 4.3: Implementation of merging of two arrays.

Chapter 5

Passing Multiple References on a Free Cluster

5.1 Introduction

In the introduction chapter we have shown that the Universe type system with ownership transfer can handle the merging of two lists in a simple way. This slightly modified method illustrates the concatenation of two list where the second list is given by a `free` reference to the first node of the list:

```
void concatenate(free Node otherHeader) {
        header.prev.next = otherHeader.next;
        otherHeader.prev.next = header;
        otherHeader.next.prev = header.prev;
        header.prev = otherHeader.prev;
}
```

If instead of merging both whole lists we want to ass only a part of the second list given by the first and the last element, the method would look as follows:

```
void concatenate(free Node first, free Node last) {
        header.prev.next = first ;
        last .next = header;
        first .prev = header.prev;
        header.prev = last;
}
```

We really do not want that *first* and *last* point to different `free` clusters but to the same `free` cluster because if we now want to pass two references to the same list we get the problem that after passing of the first parameter the second gets `any`:

```
LinkedList l1 = new LinkedList(); // {l1 | CL_any}
LinkedList l2 = new LinkedList(); // {l1 | l2 | CL_any}
...
Node header = l2.getHeader(); // {l1 | header | CL_any | l2}
l1 .concatenate(header.next, header.prev);
// after passing first argument: {l1 | l2 | CL_any header}
```

This means that we have no possibility to declare that we want to have more than one reference to the same `free` cluster as parameter for a method.

Goal. Our goal is to allow the passing of multiple references on the same `free` cluster via method invocation.

5.2 Solution

5.2.1 Declaration Syntax

`free` references can only be used for parameters and return values. Since there is always only one return value the multiple references problem can only occur with parameters. To declare that two parameters point into the same `free` cluster we use the following syntax:

1. One of the references that point into the same cluster - we call it *param* - is normally declared as `free`.
2. All other references that point into the same cluster are declared as `rep`[param].

This means that for the example above we have to change the declaration of the method *concatenate* as follows:

```
void concatenate(free Node first, rep[ first ] Node last)
```

It can be smoothly used for more than one `free` cluster too:

```
void concatenate(free Node firstList1, rep[ firstList1 ] Node lastList1,
        free Node firstList2, rep[ firstList2 ] Node lastList2)
```

The order of the parameter does not matter:

```
void concatenate(rep[firstList1 ] Node lastList1, rep[ firstList2 ] Node lastList2,
                free Node firstList1, free Node firstList2)
```

And there can be clearly more than two references that point into the same cluster:

```
void m(free T p1, rep[p1] T p2, rep[p1] T p3)
```

We use the same syntax for declaring that a parameter points into a `free` cluster declared by another parameter as well as that a parameter points into the cluster defined by a field. Thus we use the following order to discover the cluster of a parameter p declared as `rep`[x]:

- If a parameter in the same method is called x and this parameter is declared as `free` then p points into the same cluster as x.
- Else (this includes the cases when there exists a parameter x but it is not declared as `free`) we search for a field x that is declared `uniq` in the same class as the method is defined. If there exists such a field p points into the cluster defined by this field and thus stays of type `rep`[x].
- Else we get a type error.

5.2.2 Semantic

If we have multiple references to one `free` cluster the reference declared as `free` is not really externally unique anymore. We handle all these references that point into the `free` cluster like `free` references, but they are coupled together. This means that at the beginning of the method all these references are in one cluster. We illustrate this in the following example:

```
uniq T f;
peer T g;

void m(free T p1, free T p2) {
        // {p1 | p2 | f | CL_peer g}
        g = p1;
        // {p2 | f | CL_peer g | unusable p1}
        f = p2;
        // {p2 f | CL_peer g | unusable p1}
}

void n(free T p1, rep[p1] T p2) {
        // {p1 p2 | f | CL_peer g}
        g = p1;
        // {f | CL_peer g | unusable p1 p2}
        f = p2; // not allowed since p2 is unusable
}
```

Both parameters of the method m are of type `free` and could be handled independently of each other. In the case of n the two parameters are `free` references too, but they are coupled since they are in the same cluster at the beginning of the method. Thus, reading $p1$ makes not only $p1$ unusable but $p2$ too. Therefore, the second field writing in n is not allowed since $p2$ is unusable.

If two formal parameters are declared of the same `free` cluster it is not needed that the actual parameters are in the same cluster at time of method call too. Since only transferable clusters are assignable to a `free` reference we could, without problem, merge the clusters of the two parameters.

5.2.3 Order of Type Checking and Transition Rules

For each parameter of a method call we create - if the formal parameter type is not `any` - one merge operation. In addition we have to check for each parameter whether the actual parameter is assignable to the formal parameter. Since we get no operation for `any` parameters we first concentrate on all parameters and discuss the different handling of `any` parameters later.

Since the merge operation of a parameter can change the type of the later actual parameters it is important that the assignable-to check has to be done at the point in the analysis directly before the merge operation of this parameter. Thus, the order will be type checking for the first parameter, then transition rule for the first parameter, then type checking for the second parameter, then transition rule for the second parameter and so on. Therefore, in the following example the method call $m(x, y)$ is not well typed since y becomes `any` after passing x as first parameter.

```
void m(free T p1, free T p2) {}

void n() {
        T x,y;
        x = new T();
        y = x;
        // {x y}
        m(x,y);
}
```

If we apply this rule to a method with multiple references to one `free` parameter and pass as arguments two variables pointing into the same cluster we get troubles:

```
void m(free T p1, peer T p2, rep[p1] T p3) {}
```

```
    void n() {
        T x,y,z;
        x = new T();
        y = x;
        z = new T();
        // {x y | z}
        m(x, z, y);
    }
```

After passing x we merge the cluster of x with `any` and thus, y gets `any` too. But since we want to allow it in this special case we have to do the assignable-to check for y at the same point as for x. On the other hand we should consider that there possibly is a dependency between z and y too. This leads us to the necessity to change the order of the parameters (clearly the formal as well as the actual parameters). If we modify the order to $m(x, y, z)$ and m(`free` T p1, `rep`[p1] T p3, `peer` T p2) we can type check y at the same point as x and the dependency between z and y is considered since when we check the assignability for z we consider the merge operation of y.

The reordering has to be done that each group of multiple references to the same `free` cluster should be handled successively and the assignable-to check for all of them has to be done directly before the first merge operation of this group of parameters is executed. In other words we first do all assignable-to checks for a group and then all merge operations of this group before we continue with the next group.

We do this reordering only for the analysis statements. In the original abstract syntax tree we do not change the order. We will show in a later section why the reordering of the parameters does not change the correctness of a program.

any Parameters. Since no merge operation is created for formal `any` parameters we only have to look at their assignability check. The closest solution would be to check the assignability for these parameters in the order we get by our reordering like for all other parameters that do not point into a `free` cluster. Since for `any` parameters the order of the parameters matters this is not the best solution. We illustrate these circumstances by the following example:

```
class T {
    peer T p;
    uniq T f;

    void m1(any T r, peer T r2) {}

    void n(){
        // {peer p | CL_f f | unusable}
        f.m1(f, f.p);
        // after passing the first parameter: {CL_peer p | CL_f f | unusable}
        // after passing the second parameter: {CL_peer p | CL_f | unusable f}
        f = new T();
        // {CL_peer p | CL_f | unusable | f}
    }
}
```

```
class A {
    peer A p;
    uniq A f;

    void m2(peer A r, any A r2) {}

    void n(){
```

```
            // {peer p | CL_f f | unusable}
            f.m2(f.p, f);
            // after passing the first parameter: {CL_peer p | CL_f | unusable f}
            // assignability check for second parameter fails
        }
}
```

The method call $f.m1(f, f.p)$ in class T is well typed while quite the same method call in class A - we only change the order of the parameters - fails. The problem there is that f becomes unusable after passing the first parameter and thus f is not assignable to `any` anymore. According to the owner-as-modifier property it would not be a problem if we allow passing a read-only reference to an unusable variable. Thus, we want to allow the method call in the second class.

To be more flexible we check for each `any` parameter if the actual parameter is assignable to the formal parameter at the beginning of the method call before the first merge operation of the first parameter. In the case of our example above we see that f is before the method call of type `rep`[f] and therefore assignable to `any`.

Example. We use the following method to visualize the reordering:

m(`free` T f, `peer` T p, `rep`[f] T f2, `free` T g, `rep`[f] T f3, `rep` T r, `rep`[g] T g2, `any` T a)

In figure 5.1 you see which order the analysis statements of these eight parameters have. Note that we use $Merge(f)$ as short for the merge statement created in correspondence with the formal parameter f. We first have the analysis statements of the parameters p, r, and a which do not point into a `free` cluster. After we have the analysis statements for each `free` cluster for all parameters pointing into it. This means first the analysis statements for f, $f2$, and $f3$ and after these for g and $g2$.

In contrast figure 5.2 shows for each parameter before which analysis statement it should be checked whether the actual parameter is assignable to the formal parameter or not. As mentioned above we do this check for the `any` parameter a at a dummy skip analysis statement at the beginning. The two parameters p and r are checked before its own merge operation. The multiple references to one `free` cluster like f, $f2$, and $f3$ are all checked together before the analysis statement of f.

Impact of Reordering of Parameters. To show that the reordering of the parameters has no impact on the semantic of the program we have to prove that:

1. we can omit all formal parameters of type `any`
2. the reordering of the merge statements does not change the property space at the end of the method call
3. the different order of the assignability checks raises in the same cases an error as if we would check it in the declared order (except for multiple references on a `free` cluster where the assignability check in the old order would lead to unintentional type errors)

We can omit all parameters of type `any` since on the one hand they have no impact on the analysis values because they create no merge operation. On the other hand we check the assignability of these parameters always at the beginning of the method call and thus, the order does not matter.

We now show that the reordering of the merge statements does not matter for the property space or more formally that in both orders the resulting analysis value L' is the same (where M_i stands for a merge statement and L is the analysis value):

$$L \vdash_L M_1; \ldots; M_i; M_{i+1}; \ldots; M_k; L'$$

$$L \vdash_L M_1; \ldots; M_{i+1}; M_i; \ldots; M_k; L'$$

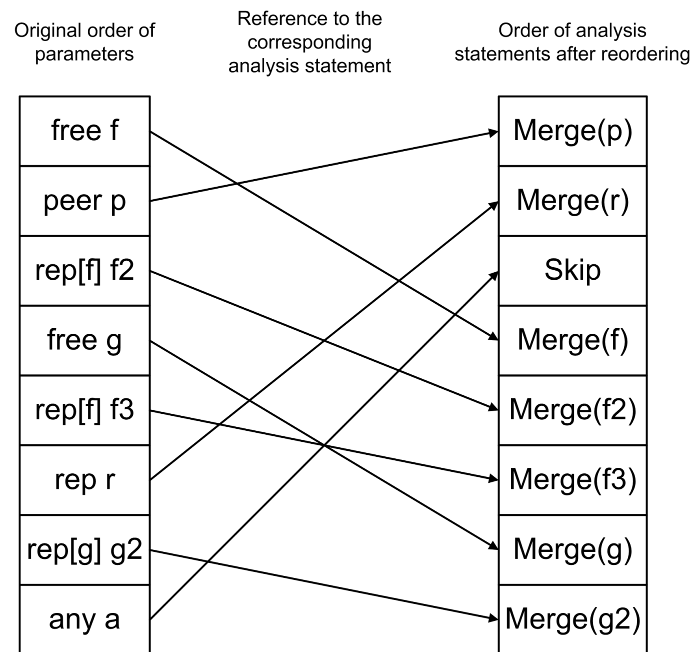

Figure 5.1: Reordering of analysis statements of parameters.

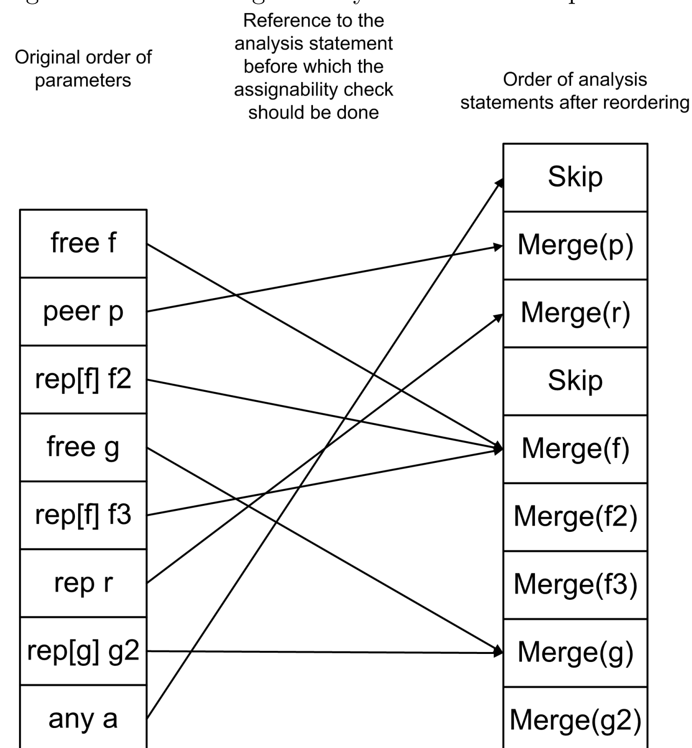

Figure 5.2: Analysis statements before which each parameter should be type checked.

Or in other words that in the following both cases L' is the same (while Cl_i represents the cluster the corresponding variable belongs to):

$$\ldots; L \vdash_L Merge(Cl_1, Cl_2); Merge(Cl_3, Cl_4); L' \ldots;$$

$$\ldots; L \vdash_L Merge(Cl_3, Cl_4); Merge(Cl_1, Cl_2); L' \ldots;$$

We now look at different cases. If $Cl_1 = Cl_2$ or $Cl_3 = Cl_4$ the corresponding merge operation has no impact. Else if all four clusters are different, the order of the merge operations does not matter too. The most interesting cases occurs if two clusters in both merge operations are the same like $Cl_1 = Cl_3$. Then we get the following situation:

$$L \vdash_L Merge(Cl_1, Cl_2); Merge(Cl_1, Cl_4); L'$$

$$L \vdash_L Merge(Cl_1, Cl_4); Merge(Cl_1, Cl_2); L'$$

We divide this once again in different subcases:

- If all clusters Cl_i are `rep` clusters then their merge operations perform only union on sets. Since the union operation is commutative the order of the merge operations does not matter.
- If Cl_2 is the `peer` or `any` cluster we have to consider that beside of the union operation `rep` fields become unusable. In the first merge operation all fields of cluster Cl_1 become unusable. In the second operation all fields of cluster Cl_4 become unusable too. In the reverse order during the first operation no field becomes unusable. When we perform the second merge operation, all fields in cluster Cl_1 - and those in cluster Cl_4 too since $Cl_1 = Cl_4$ after the first merge operation - becomes unusable. Since in both cases the same fields get unusable we have the same analysis value after the two operations.
- If Cl_1 is the `peer` or `any` cluster in the first order, all fields of cluster Cl_2 and Cl_4 become unusable. In the second order all fields of cluster Cl_4 and Cl_2 becomes unusable too, only in reverse order. Once again the order of the merge statements has no effect on final analysis value.

This semi-formally prove shows that the order of the merge statements has no impact on the property space after the last merge statement.

As a last point we have to show that the reordering has no impact whether a method call is well typed or not. For easier proving we omit the `any` parameters since they are always checked at the beginning. In addition we include for each group of references into the same `free` cluster only the first reference because we have shown above why all other references of the same group have to be checked at the same point. If we look at two different parameters x and y the type check and the merging operations have the following order:

$$check(x)$$

$$merge(x, Cl_1)$$

$$check(y)$$

$$merge(y, Cl_2)$$

We now want to prove that this method call $m(x, y)$ will only be accepted iff in the reverse order:

$$check(y)$$

$$merge(y, Cl_2)$$

$$check(x)$$

$$merge(x, Cl_1)$$

$m(y, x)$ is accepted too. If x and y are not in the same cluster at the beginning, the first merge statement has no impact on the result of the second check. Thus, we have to concentrate on the case where x and y are in the same cluster at the beginning. The only reordering we do is moving the `free` parameters. If we handle x as `free` parameter we get the following situation:

$$\{\text{x y} \mid \text{any}\}$$
$$check(x)$$
$$merge(x, any)$$
$$\{\text{any x y}\}$$
$$check(y)$$
$$merge(y, Cl_2)$$
$$\{\text{any x y}\}$$

and reordered:

$$\{\text{x y} \mid \text{any}\}$$
$$check(y)$$
$$merge(y, Cl_2)$$
$$\{\text{x y} \mid \text{any}\}$$
$$check(x)$$
$$merge(x, any)$$
$$\{\text{x y any}\}$$

As mentioned above all parameters with formal parameter `any` are handled separated. Thus, in the first order $check(y)$ will fail since the actual parameter is `any` and it would not be assignable to a formal parameter that is not `any`. The second order is a little bit more complicated. We have to distinguish two different cases:

- If Cl_2 is a transferable cluster or in other words the formal parameter of y is `rep` $\langle Cl_{tr} \rangle$, both assignable checks will run without problems. But according to our type rules if the formal parameter is `rep` the receiver of the method call should be `this` and thus we have to perform $InvTest$ after the method call. Since Cl_2 is defined by a field it contains at beginning of the method call at least this field. With the second merge operation the cluster Cl_2 is merged with the `any` cluster and thus the defining field becomes unusable. Hence the invariant test will fail.
- If Cl_2 is a non-transferable cluster or in other words the formal parameter of y is `rep` $\langle Cl_{nonTr} \rangle$ or `peer`, x will be in a non-transferable cluster before its check. Thus, the assignable-to check will fail.

This means that we accept with reordering the same programs as we would with the original order. Note that the only difference is that the given error message can vary.

We now have shown that for the change of two subsequent parameters the reordering has no impact on the property space as well as whether the method call is well typed. Since we get each arbitrary permutation by a sequence of such changes between two parameters the same property holds for any order of the parameters.

5.2.4 Summary

Summarized passing multiple references to one cluster via method invocation is handled as follows:

- If a parameter p is declared as `rep`[param] where *param* is a `free` parameter in the same method, this means that p points into the same `free` cluster as *param* at beginning of the method.
- It is not needed that the corresponding two actual parameters are in the same cluster before the method call, they should only point both into a transferable cluster.
- For checking the type rules the order of the parameters is changed.

Chapter 6

Formalization

This chapter presents the formalization of the shown type system for a simplistic toy language. The static data flow analysis used in this chapter is fully covered in the subsequent chapter.

This chapter relies on the corresponding chapter "Formalization" in [12]. To make understanding easier we use the same syntax to describe the formalization. The parts of the formalization, which do not have changed in comparison to the old version, are only mentioned shortly. For further reading we refer to [12].

6.1 Toy Language Syntax

We use a simply, minimalistic toy language to give the formalization of the type system. We will describe it with the following syntactic categories:

P	$\in$	**TProg**	programs
$CDecl$	$\in$	**TCDecl**	class declarations
$MDecl$	$\in$	**TMDecl**	method declarations
W	$\in$	**TPure**	purity modifiers
S	$\in$	**TStmt**	statements
$T_f, T_p, T_r, T_l, T_n, T_c, T_a$	$\in$	**TType**	field types, formal parameter types, return value types, local variable types, object creation types, types in cast statements, and array element types
m_f, m_p, m_r, m_c, m_a	$\in$	**TMod**	Universe modifiers for field types, formal parameter types, return value types, types in cast statements, and array element types

Note that - in contrary to the old version - we do not have modifiers for local variable types and object creation types. Another difference to the old version is that we add types and modifiers for array elements to the language.

Further, we make use of the following meta variables and shortcuts as in [12]:

C, D	$\in$	**TClass**	class names (including Object)
mt	$\in$	**TMethod**	method names
f	$\in$	**TField**	field names (including result and formal parameter names)
p	$\in$	**TParam** $\subseteq$ **TField**	formal parameter names
x, y, z	$\in$	**TLoc**	local variable names (including `this`)

$\overline{X}$	is a shortcut for	$X_1 \cdots X_k$	where X stands for $CDecl$ or $MDecl$
$\overline{XY}$	is a shortcut for	$X_1Y_1; \cdots ; X_kY_k;$	where $X \in$ **TType** and $Y \in$ (**TField** $\cup$ **TLoc**)
$\overline{[x]}$	is a shortcut for	$[x_1] \cdots [x_k]$	where $x \in$ **TLoc**

Note that we handle parameters and result values like fields and not like local variables as it was done in the existing system. The syntax of the toy language is given in table 6.1. Let us comment some aspects that differ from the old solution:

- For local variable declaration and new expression we expect only a class declaration but no ownership modifier.
- For array handling we allow array reading, array writing as well as array creation expression.
- We allow casting to the cluster a local variable belongs to. It should be checked that the variable is of type `rep` at the point of the casting.

The allowed usage of Universe modifiers is summarized in the following table (where f represents a field declared `uniq` in the enclosing class, x is a local variable with inferred type `rep`, and p is a parameter declared `uniq` in the enclosing method):

	no modifier	any	peer	rep	rep[f]	rep[x]	rep[p]	uniq	free
Field declaration	-	yes	yes	yes	yes	-	-	yes	-
Formal parameters	-	yes	yes	yes	yes	-	yes	-	yes
Return values	-	yes	yes	yes	yes	-	-	-	yes
Local variable declarations	yes	-	-	-	-	-	-	-	-
New statements	yes	-	-	-	-	-	-	-	-
Cast statements	-	yes	yes	yes	yes	yes	-	-	-
Array element	-	yes	yes	-	-	-	-	-	-

6.2 Clusters

The set **TClust** represent the domain for possible cluster names to identify clusters. In addition to the clusters declared by the keyword `uniq` and the `this` Cluster Cl_{this} - which is now called

$$
\begin{array}{lcl}
P & ::= & \overline{CDecl} \\
CDecl & ::= & class\ C\ extends\ D\ \{\overline{T_f\ f};\ \overline{MDecl}\} \\
MDecl & ::= & W\ T_r\ mt(T_p\ p)\ \{\overline{T_l\ y}\ S\} \\
W & ::= & pure \\
 & | & nonpure \\
S & ::= & x = y \\
 & | & x = null \\
 & | & x = new\ T_n \\
 & | & x = new\ T_n\overline{[x]} \\
 & | & x = y.f \\
 & | & x.f = y \\
 & | & x = y[z] \\
 & | & x[z] = y \\
 & | & x = y.mt(z) \\
 & | & x = (T_c)\ y \\
 & | & S_1; S_2 \\
T_f & ::= & m_f\ C \mid m_f\ T_a \\
T_l & ::= & C \mid T_a \\
T_a & ::= & m_a\ C[] \\
T_n & ::= & C \\
T_i & ::= & m_i\ C \quad ,\ for\ i \in \{p, r, c\} \\
m_f & ::= & \texttt{any} \mid \texttt{peer} \mid \texttt{rep}\,[f] \mid \texttt{uniq} \mid \texttt{rep} \\
m_p & ::= & \texttt{any} \mid \texttt{peer} \mid \texttt{rep}\,[f] \mid \texttt{free} \mid \texttt{rep} \\
m_r & ::= & m_p \\
m_c & ::= & \texttt{any} \mid \texttt{peer} \mid \texttt{rep} \mid \texttt{rep}\,[x] \mid \texttt{rep}\,[f] \\
m_a & ::= & \texttt{any} \mid \texttt{peer}
\end{array}
$$

Table 6.1: Syntax of the toy language.

Cl_C for class C - we insert new cluster names for each superclass of the current class. This is a conclusion from the subclass separation.

The function transCls: **TClass** $\rightarrow 2^{TClust}$ yields the transferable clusters defined by the class C (where definedCl: **TField** $\rightarrow$ **TCluster** returns the name of the cluster defined by a given field name and the symbol _ denotes a "don't care" placeholder):

$$\frac{}{transCls(Object) = \emptyset}$$

$$\frac{class\ C\ extends\ D\ \{\ ...;\ uniq\ _\ f;\ ...\ _\ \}}{transCls(C) = \{definedCl(f)\} \cup transCls(D)}$$

While the functions nonTransCls: **TClass** $\rightarrow 2^{TClust}$ yields the non-transferable clusters defined by the class C:

$$\frac{}{nonTransCls(Object) = \{Cl_{Object}, Cl_{any}, Cl_{peer}\}}$$

$$\frac{class\ C\ extends\ D\ \{...\}}{nonTransCls(C) = \{Cl_C\} \cup nonTransCls(D)}$$

All clusters defined by the class C are yield by definedCls: **TClass** $\rightarrow 2^{TClust}$:

$$definedCls(C) = transCls(C) \cup nonTransCls(C)$$

Note that we rename the cluster Cl_{this} to Cl_C where C is the current class. We assume that **TClust** is disjoint from each of **TClass**, **TMethod**, **TField**, and **TLoc** so that globally unique identifiers are guaranteed.

We distinguish between transferable and non-transferable clusters. All clusters derived from a class - as Cl_{Object} and Cl_C - are non-transferable whereas the clusters derived from a field like Cl_f are transferable. Subsequently we use Cl_{tr} to denote a transferable cluster and Cl_{nonTr} to mark a non-transferable cluster:

$$Cl_{tr} \in transCls(C)$$
$$Cl_{nonTr} \in nonTransCls(C)$$

6.3 Universe Modifier Translation

In the toy language we use the following Universe modifiers:

$$\mathbf{TMod} = \{\texttt{this}, \texttt{any}, \texttt{peer}, \texttt{rep}, \texttt{uniq}, \texttt{free}\} \cup \{\texttt{rep}\,[f] : f \in \mathbf{TField}\} \cup \{\texttt{rep}\,[x] : x \in \mathbf{TLoc}\}$$

with `this` as Universe modifier for the `this` reference. Remember that parameters are included in the set **TField**.

These modifiers correspond to the declared types. For further handling we have to use another set of modifiers since for example a `rep`[param] parameter should be handled like a `free` reference and a `uniq` field is treat like `rep`. Thus, we do a translation for each Universe modifier in **TMod** to one in the following set:

$$\mathbf{CMod} = \{\texttt{this}, \texttt{any}, \texttt{peer}, \texttt{free}\} \cup \{\texttt{rep}\,\langle Cl\rangle : Cl \in \mathbf{TClust}\} \cup$$

$$\{\texttt{rep}\,\langle x\rangle : x \in \textbf{TLoc}\} \cup \{\texttt{rep}\,\langle f\rangle : f \in \textbf{TField}\} \cup \{\texttt{free}\,\langle p\rangle : p \in \textbf{TParam}\}$$

`this`	$\mapsto$	`this`
`any`	$\mapsto$	`any`
`peer`	$\mapsto$	`peer`
`rep`	$\mapsto$	$\texttt{rep}\,\langle Cl_C\rangle$, where C is the enclosing class
`uniq`	$\mapsto$	$\texttt{rep}\,\langle definedCl(f)\rangle$, where f is the declared field
`free`	$\mapsto$	`free`
$\texttt{rep}\,[f] \neq m_c$	$\mapsto$	$\texttt{rep}\,\langle definedCl(f)\rangle$, where f must be declared `uniq` in the enclosing class
$\texttt{rep}\,[f] = m_c$	$\mapsto$	$\texttt{rep}\,\langle f\rangle$, where f must be declared `rep` in the enclosing class
$\texttt{rep}\,[p]$	$\mapsto$	$\texttt{free}\,\langle p\rangle$, where p must be declared `free` in the enclosing method
$\texttt{rep}\,[x]$	$\mapsto$	$\texttt{rep}\,\langle x\rangle$, where x must be a declared local variable in the enclosing method

We call these modifiers the core Universe modifiers. To distinguish these two modifier type we use in combination with `rep` '[]'-brackets for declaration modifiers and '<>'-brackets for the core modifiers. Subsequently we will always use the core modifiers. In addition we use the following shortcut:

$$\texttt{rep} = \texttt{rep}\,\langle Cl\rangle\,, for\ any\ arbitrary\ cluster\ Cl \in \textbf{TClust}$$

Whenever only `free` is mentioned we include all variables with core modifier $\texttt{free}\,\langle p\rangle$ too. Notice that there are no translations for local variables since they have no declared modifier.

6.4 Lookup Functions

The lookup functions for static information can be taken from the old version. These are:

- The function
$$fields : \textbf{TClass} \rightarrow 2^{\textbf{TField}}$$
yields the identifiers declared in or inherited by a given class.

- The function
$$fType : \textbf{TClass} \times \textbf{TField} \rightarrow \textbf{TType}$$
yields the type of a field as declared in the given class.

- The function
$$mType : \textbf{TClass} \times \textbf{TMethod} \rightarrow \textbf{TPure} \times \textbf{TType} \times \textbf{TType}$$
yields the signature of a given method as declared in the specified class.

- The function
$$mLoc : \textbf{TClass} \times \textbf{TMethod} \rightarrow 2^{\textbf{TLoc}}$$
yields the names of the local variables of a method.

- The function
$$\Delta : \textbf{TClass} \times \textbf{TMethod} \rightarrow (\textbf{TLoc} \rightarrow \textbf{TType})$$
yields the static declaration environment for a given method in the specified class.

We need one more function for handling arrays:

- The function

$$elemType : \mathbf{TClass} \times \mathbf{TMethod} \times \mathbf{TLoc} \rightarrow \mathbf{TType}$$

yields the declared modifier of array elements.

$$\frac{class\ C\ extends\ _\ \{_\ \ldots\ _\ _\ mt(_\ _)\ \{\ldots\ T_a\ ar\ldots\}\ \ldots\ \}}{elemType(C, mt, ar) = T_a}$$

6.4.1 Type Projection Functions

We use two projection functions to access the modifier an class name of a type $T = m\ C$:

- The function

$$mod : \mathbf{TType} \rightarrow \mathbf{CMod}$$

yields the core modifier of a given type.

- The function

$$class : \mathbf{TType} \rightarrow \mathbf{CClass}$$

yields the class name of a given type.

6.5 Type Combinator

We define the type combinator which is used to determine the type of transitive access for the core modifiers as follows:

$\triangleright_U$	any	peer	rep<Cl>	free
this	any	peer	rep $\langle Cl \rangle$	free
any	any	any	any	free
peer	any	peer	any	free
rep $\langle Cl \rangle$	any	rep $\langle Cl \rangle$	any	free

Table 6.2: Core Universe modifier combinator. The left-most cell of the rows means the first argument, the top-most cell of the columns means the second argument.

Inside of a method we handle the parameters declared as `free` like `rep` $\langle Cl_{tr} \rangle$. Thus, we never have `free` on the left hand side of an transitive access and can therefore omit the corresponding line in the table.

6.6 Assignable-To Relations

In our type system the subtype relation and the assignable-to relation differs. Because of ownership transfer we can assign more than only subtypes. The subtype relation outlined in figure 6.1 follows by a being-less-specific-than point.

One type $m_1\ C_1$ is assignable to another type $m_2\ C_2$ iff C_1 is the same class as C_2 or is a subclass of C_2 (like in Java) and if modifier m_1 is assignable to modifier m_2 according to the assignable-to-relation shown in figure 6.2.

We should additionally mention that two non-transferable `rep` clusters are not assignable to each other if they are not the same. In other words the following condition holds:

$$\texttt{rep}\,\langle Cl_{nonTr1} \rangle\ \leq_A\ \texttt{rep}\,\langle Cl_{nonTr2} \rangle\ \ iff\ Cl_{nonTr1} = Cl_{nonTr2}$$

All other types are always assignable to the same type:

$$\texttt{rep}\,\langle Cl_{tr1} \rangle\ \leq_A\ \texttt{rep}\,\langle Cl_{tr2} \rangle$$

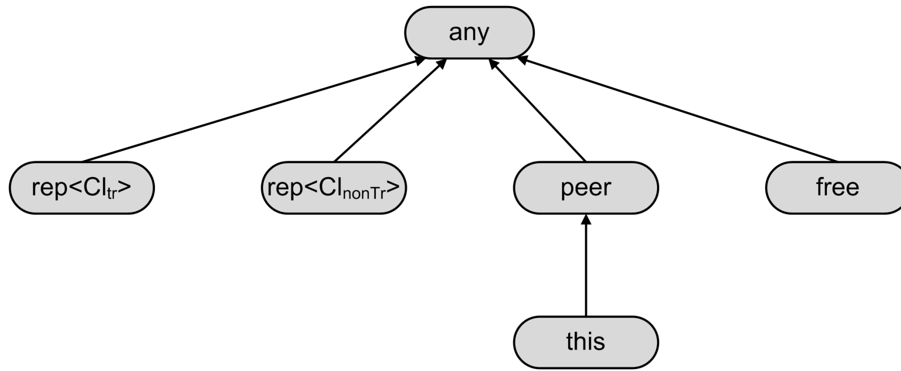

Figure 6.1: Subtype relation.

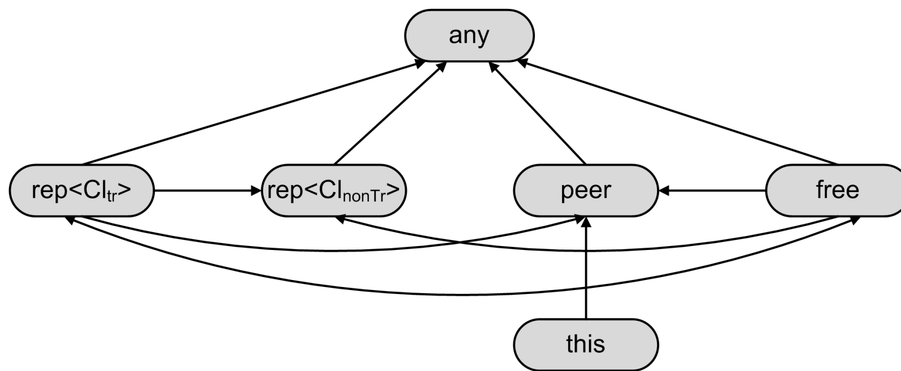

Figure 6.2: Assignable-to relation. Cl_{tr} means a transferable cluster whereas Cl_{nonTr} is a non-transferable cluster.

The assignable-to relation for arrays is illustrated in figure 4.2. In general the assignable-to relation should only be used if there is a field or a parameter on the left hand side of the assignment since an assignment to a local variable is always correct regardless of which type is on the right hand side.

6.7 Ternary Logic

For the static data flow analysis we use a ternary logic with the value set

$$\mathbb{T} := \{0, \frac{1}{2}, 1\}$$

.

When we query the data flow analysis it deliver an answer $a \in \mathbb{T}$ with the following interpretation:

$a = 0$	$\Rightarrow$	The property definitely does not hold in all possible executions of the program.
$a = 1$	$\Rightarrow$	The property definitely holds in all possible executions of the program.
$a = \frac{1}{2}$	$\Rightarrow$	The property can hold in some executions, but it also cannot hold in other executions of the program. We do not know it.

6.8 Static Data Flow Analysis - Introduction

6.8.1 Analysis Values and Queries

An analysis value represents which variable points into in which cluster at a specific program point. $\mathcal{L}$ means the property space or the analysis universe which is formed by all analysis values.

The special analysis value $\imath \in \mathcal{L}$ symbolizes the initial value that stands at the beginning of the program. Informally it can be described as follows:

- Each field points into its declared cluster.
- Each local variable points into a new own cluster.
- The parameters of type `free` $\langle p \rangle$ as well as the parameter p itself point into the same cluster. All other parameters point to their declared cluster.

Like in the old version we offer the following two query functions to the analysis values:

$$isUnusable: \mathcal{L} \times fields(C) \rightarrow \mathbb{T}$$

returns whether a analysis variable is unusable. Consider that local variables cannot become unusable hence the function is not used for local variables.

The second function is:

$$pointsInto: \mathcal{L} \times (mLoc(C, mt) \cup fields(C)) \times definedCls(C) \rightarrow \mathbb{T}$$

checks whether an analysis variable points into a given cluster.

We introduce some more queries to receive type information about local variables and fields from analysis values. Since we work with ternary logic a simple type modifier query is not possible because the answer can be ”possibly `peer`” and ”possibly `any`”. Thus we have to implement queries for the three types `peer`, `rep`, and `any`.

The function

$$isPeer: \mathcal{L} \times mLoc(C, mt) \cup fields(C) \rightarrow \mathbb{T}$$

checks whether a given field, parameter value or a local variable is of type `peer` at this program point. In case of fields and parameter values this query takes the information from the static declared types. For local variables the answer is taken from the analysis values.

Analogously, the function

$$isAny: \mathcal{L} \times mLoc(C, mt) \cup fields(C) \rightarrow \mathbb{T}$$

returns whether a given field, parameter value or a local variable is of type `any` at this program point. For fields and parameter values the static declared type is taken while for local variable we use the current type in the analysis value.

Furthermore, the function

$$isRep: \mathcal{L} \times mLoc(C, mt) \cup fields(C) \rightarrow \mathbb{T}$$

tests whether a given field, parameter value or a local variable is of type `rep` at this program point. We check this with a *pointsInto*-query for local variables as well as for fields and parameter variables.

6.8.2 Analysis Transition Functions

The analysis transition operates on the property space $\mathcal{L}$ and represents the effect a certain entity has on an analysis value $L \in \mathcal{L}$. We use the following transition functions:

- $Merge(L, x, y)$: x and all variables in the same cluster as x are marked as pointing in the same cluster as y.
- $Merge(L, x, Cl)$: x and all variables in the same cluster as x are marked as pointing in the cluster Cl.
- $Move(L, x, y)$: x is marked as pointing into the same cluster as y.
- $Move(L, x, Cl)$: x is marked as pointing into the cluster Cl.
- $New(L, x)$: x is marked as pointing into a new cluster. x is the sole variable that points into this new cluster.
- $InvRestore(L, x)$: if x is of type `peer`, all local variables pointing into the same transferable cluster as a fields are marked as pointing into the cluster Cl_{any} and the function $RestoreFields(L)$ is executed.
- $RestoreFields(L)$: if two fields of same declared type are in different clusters they are marked as pointing into the same cluster.

These transition function will be defined formally in chapter 7. Note that we do not need the functions *Consume* and *ConsumeLocals* introduced in the old version anymore.

6.9 Analysis Value Transition Rules

The analysis value transition rules describe the effect that each statement of the toy language has in terms of the analysis values. The rules have the following structure:

$$\Gamma; L \vdash e; L'$$

which means that in a declaration environment Γ partition change from L to L' after evaluation expression e. The analysis value transition rules are listed in figure 6.3 and 6.4.

We use some shortcuts for easier understandability:

- m_f - declared type modifier of field f:

  $$m_f = mod(fType(C, f))$$

  where C is the class where f is declared

- m_{yElem} - declared type modifier of the elements of array y:

  $$m_{yElem} = mod(elemType(C, mt, y))$$

  where y is declared in the method mt of class C

- mt_p - type modifier of method's mt input parameter:

  $$mt_p = mod(T_p) \textit{ and } (W, T_p, T_r) = mType(C, mt)$$

  where C is the class where mt is declared

- mt_{res} - type modifier of method's mt result:

  $$mt_{res} = mod(T_r) \textit{ and } (W, T_p, T_r) = mType(C, mt)$$

  where C is the class where mt is declared

- mi_x - the inferred type modifier of x

Further we use sometimes a modifier m instead of an analysis variable as argument of the *move* or *merge* operations like $merge(L, x, m)$. To get the corresponding analysis variable we replace m by the cluster marker if it specifies a non-transferable cluster. Otherwise if m specifies a transferable cluster, we replace m with the defining field of this cluster.

Moreover sometimes the type combinator is used inside a *move* or *merge* operation like $merge(L, x, y \triangleright_U m_f)$. How the combinator and its result is mapped to an analysis variable is shown in section 8.3.4.

new Expression. Parameters of a `new` expression $x = new\ T(p1, p2)$ are handled like parameters of a method call $x.T(p1, p2)$.

return Statement. The `return` statement *return* x is handled like a method call $this.m(x)$ where the formal parameter of the imaginary method m has the same type as the return type of the current method. Additionally we have to add an *exit* statement at the end considering that we exit the method.

Method Invocation with many Parameters. If we have a method call with more than one parameter in the rule [L-PRE-INVK] we first have one $RestoreFields$ operation and after we create for each parameter - if the formal parameter is not `any` - its *merge* operation. These operations should be in such an order that the operations of the formal parameters which point into the same `free` cluster are one after the other.

$$\frac{L' = move(L, x, y)}{\Gamma; L \vdash_L x = y; L'} \quad \text{[L-ASSIGN]}$$

$$\frac{\begin{array}{c} L' = if(y = \texttt{this} \wedge m_f = \texttt{rep})\ then\ move(L, x, f) \\ else\ move(L, x, y \rhd_U m_f) \end{array}}{\Gamma; L \vdash_L x = y.f; L'} \quad \text{[L-FD-RD]}$$

$$\frac{\begin{array}{c} L' = if(y = \texttt{this} \wedge m_f = \texttt{rep}\, \langle Cl_{tr} \rangle)\ then\ move(L, f, x) \\ else\ if(m_f \neq \texttt{any})\ merge(L, x, y \rhd_U m_f)\ else\ L \end{array}}{\Gamma; L \vdash_L y.f = x; L'} \quad \text{[L-FD-WR]}$$

$$\frac{\begin{array}{c} if(m_{yElem} = peer)\ then\ move(L, x, y) \\ else\ move(L, x, Cl_{any}) \end{array}}{\Gamma; L \vdash_L x = y[z]; L'} \quad \text{[L-ARR-RD]}$$

$$\frac{if(m_{yElem} \neq \texttt{any})\ merge(L, x, y)\ else\ L}{\Gamma; L \vdash_L y[z] = x; L'} \quad \text{[L-ARR-WR]}$$

$$\frac{\begin{array}{c} RestoreFields() \\ L' = if(mt_p = \texttt{any})\ then\ L \\ else\ if(mt_p = \texttt{free})\ then\ merge(L, z, Cl_{any}) \\ else\ merge(L, z, y \rhd_U mt_p) \end{array}}{\Gamma; L \vdash_L y.mt(z); L'} \quad \text{[L-PRE-INVK]}$$

$$\frac{\begin{array}{c} \Gamma; L \vdash y.mt(z); L_0 \\ L' = InvRestore(L_0, y) \\ L'' = if(mt_{res} = \texttt{free})\ then\ new(L', x) \\ else\ move(L', x, y \rhd_U mt_{res}) \end{array}}{\Gamma; L \vdash_L x = y.mt(z); L''} \quad \text{[L-INVK]}$$

$$\frac{L' = new(L, x)}{\Gamma; L \vdash_L x = new\ T(); L'} \quad \text{[L-NEW]}$$

Figure 6.3: Analysis Value Transition Rules, part 1.

$$\frac{L' = new(L, x)}{\Gamma; L \vdash_L x = new\ T\overline{[y]}; L'} \quad \text{[L-NEW-ARR]}$$

$$\frac{L' = new(L, x)}{\Gamma; L \vdash_L x = null\ ; L'} \quad \text{[L-NULL]}$$

$$\frac{\begin{array}{c} L' = if(T = (m\ C))\ then\ merge(L, y, m) \\ else\ if(T = (\texttt{rep}\,\langle z\rangle\ \ C))\ then\ merge(L, y, z) \\ else\ if(T = (\texttt{rep}\,\langle f\rangle\ \ C))\ then\ merge(L, y, f) \\ else\ L \\ L'' = if(T = (m\ C))\ then\ move(L', x, m) \\ else\ if(T = (\texttt{rep}\,\langle z\rangle\ \ C))\ then\ move(L', x, z) \\ else\ if(T = (\texttt{rep}\,\langle f\rangle\ \ C))\ then\ move(L', x, f) \\ else\ move(L', x, y) \end{array}}{\Gamma; L \vdash_L x = (T)y\ ; L''} \quad \text{[L-CAST]}$$

$$\frac{\begin{array}{c} RestoreFields() \\ L' = if(mt_r = \texttt{any})\ then\ L \\ else\ if(mt_r = \texttt{free})\ then\ merge(L, z, Cl_{any}) \\ else\ merge(L, z, mt_r) \end{array}}{\Gamma; L \vdash_L return\ z; L'} \quad \text{[L-RETURN]}$$

$$\frac{\Gamma; L \vdash S_1\ ; L' \quad \Gamma; L' \vdash S_2\ ; L''}{\Gamma; L \vdash_L S_1; S_2\ ; L''} \quad \text{[L-SEQ]}$$

Figure 6.4: Analysis Value Transition Rules, part 2.

6.10 Type Rules

Type rules has the next structure:

$$\Gamma; L \vdash e$$

It means that in a declaration environment Γ and partition L expression e is well-typed. We assume that the Java types are correct and expression is well-typed according to the standard Java type rules. In addition we do not allow assignment to `this`.

The type rules for the different statements are given in figure 6.5 and 6.6, while the type rules for method and class declaration as well as for programs are presented in figure 6.7. $\imath$ means the initial analysis value.

Method Invocation with many Parameters. If we have a method call with more than one parameter the assignability check in the rule [T-INVK] has for the `free` and `free` $\langle p \rangle$ parameters to be done before the first *merge* operation of all parameters pointing into the same `free` cluster. All other parameters have to be checked before its own *merge* operation.

$$\frac{}{\Gamma; L \vdash x = y} \quad \text{[T-ASSIGN]}$$

$$\frac{y = \texttt{this} \Rightarrow isUnusable(L, f) = 0}{\Gamma; L \vdash x = y.f} \quad \text{[T-FD-RD]}$$

$$\frac{\begin{array}{c} mi_y \neq \texttt{any} \\ m_f = \texttt{rep}\,\langle_\rangle \Rightarrow y = \texttt{this} \\ (mi_x \leq_A mi_y \rhd_U m_f) \vee \\ (mi_y = \texttt{rep}\,\langle Cl_{tr}\rangle \wedge\ m_f = \texttt{peer} \wedge\ (mi_x = \texttt{peer} \vee mi_x = \texttt{rep}\,\langle Cl_{nonTr}\rangle)) \end{array}}{\Gamma; L \vdash y.f = x} \quad \text{[T-FD-WR]}$$

$$\frac{}{\Gamma; L \vdash x = y[z]} \quad \text{[T-ARR-RD]}$$

$$\frac{\begin{array}{c} mi_y \neq \texttt{any} \\ (mi_x \leq_A mi_y \rhd_U m_{yElem}) \vee \\ (mi_y = \texttt{rep}\,\langle Cl_{tr}\rangle \wedge\ m_{yElem} = \texttt{peer} \wedge\ (mi_x = \texttt{peer} \vee mi_x = \texttt{rep}\,\langle Cl_{nonTr}\rangle)) \end{array}}{\Gamma; L \vdash y[z] = x} \quad \text{[T-ARR-WR]}$$

$$\frac{\begin{array}{c} (mi_z \leq_A mi_y \rhd_U mt_p) \vee \\ (mi_y = \texttt{rep}\,\langle Cl_{tr}\rangle \wedge\ mt_p = \texttt{peer} \wedge\ (mi_z = \texttt{peer} \vee mi_z = \texttt{rep}\,\langle Cl_{nonTr}\rangle)) \\ mt_p = \texttt{rep} \Rightarrow y = \texttt{this} \vee mi_z = \texttt{rep} \\ \Gamma; L \vdash y.mt(z); L' \\ mi_y \in \{\texttt{peer},\ \texttt{this}\} \Rightarrow InvTest(L') \end{array}}{\Gamma; L \vdash x = y.mt(z)} \quad \text{[T-INVK]}$$

$$\frac{}{\Gamma; L \vdash x = new\ T()} \quad \text{[T-NEW]}$$

$$\frac{}{\Gamma; L \vdash x = new\ T\overline{[y]};} \quad \text{[T-NEW-ARR]}$$

Figure 6.5: Type Rules for statements, part 1.

$$\frac{}{\Gamma; L \vdash x = null} \quad \text{[T-NULL]}$$

$$\frac{\begin{array}{c} mi_y \; = \texttt{any} \; \vee \; mi_y \leq_A T \\ T = (\texttt{rep}\,\langle z\rangle \;\; C) \Rightarrow mi_z = \texttt{rep} \\ T = (\texttt{rep}\,\langle f\rangle \;\; C) \Rightarrow (m_f = \texttt{rep} \wedge isUnusable(L, f) = 0) \end{array}}{\Gamma; L \vdash x = (T)y \;; L'} \quad \text{[T-CAST]}$$

$$\frac{\begin{array}{c} mi_z \leq_A mt_r \\ \Gamma; L \vdash return \; z; L' \\ InvTest(L') \end{array}}{\Gamma; L \vdash return \; z;} \quad \text{[T-RETURN]}$$

$$\frac{\Gamma; L \vdash S_1; \quad \Gamma; L \vdash_L S_1 \;; L' \quad \Gamma; L' \vdash S_2}{\Gamma; L \vdash S_1; S_2} \quad \text{[T-SEQ]}$$

Figure 6.6: Type Rules for statements, part 2.

$$\frac{\begin{array}{c} if(W = \texttt{pure}) \; then \; T_r \; and \; T_p \in \{\texttt{any}, \texttt{free}\} \\ \Delta(C, m); i \vdash S \\ \Delta(C, m); i \vdash_L S; L \\ InvTest(L) \\ isUnusable(L, result) = 0 \end{array}}{C \vdash_M W \; T_r \; m(T_p \; x)\{\overline{T_l \; y} \; S\}} \quad \text{[T-MDECL]}$$

$$\frac{C \vdash_M M_1; \quad \ldots \quad C \vdash_M M_k;}{\vdash_C class \; C \; extends \; D \; \{\overline{T_f \; f} \; \overline{M}\}} \quad \text{[T-CDECL]}$$

$$\frac{\vdash_C CDecl_1; \quad \ldots \quad \vdash_C CDecl_k;}{\vdash_P \{\overline{CDecl}\}} \quad \text{[T-PROGRAMM]}$$

Figure 6.7: Type Rules for method declaration, class declaration and programs.

Chapter 7

Data Flow Analysis

This chapter describes the techniques used for the static data flow analysis. It relies on the corresponding chapter "Data Flow Analysis"' in [12]. In this chapter we will only list some few points that change in comparison with the old solution. For further understanding we refer to [12].

7.1 Syntax of the Analysis Language

The static data flow analysis does neither operate directly on the Java source, nor on the presented toy language. Its target language is an additional abstract language called the "analysis language". All operations which are not relevant for the analysis are removed from the analysis language. Thus we get a reduced amount of code.

There is only one syntactic category describing the analysis statements:

$$S \in \mathbf{AStmt} : \textit{analysis statements}$$

We use in addition a set of variables **AVar** and a set of label **ALab** for labeled statements:

$$x, y \in \mathbf{AVar} : \textit{analysis variable names}$$

$$l \in \mathbf{ALab} : \textit{analysis statement labels}$$

The syntax of the analysis language stays almost like presented in the corresponding section in [12]. First, there are elementary statements which perform a certain operation on the analysis values. They correspond to the operations used in the transition function.

$$\begin{aligned} S ::= &\ skip \\ &\mid merge(x, y) \\ &\mid new(x) \\ &\mid move(x, y) \\ &\mid restoreFields \\ &\mid invRestore(x) \\ &\mid \ldots \end{aligned}$$

Note that the two statements $consume(x)$ and $consumeLocals$ of the existing type system are not needed anymore. Instead we use the two new analysis statements $restoreFields$ and $invRestore(x)$.

All following statements are taken from the existing system. Thus, we only present them shortly. For deeper understanding we refer to [12]. There exist some elementary statements which represent a change in the control flow:

$$
\begin{aligned}
S ::= \ & \dots \\
& \mid break \mid break\ l \\
& \mid continue \mid continue\ l \\
& \mid exit \\
& \mid \dots
\end{aligned}
$$

Finally we have some composite statements which have other statements as children:

$$
\begin{aligned}
S ::= \ & \dots \\
& \mid S_1; S_2 \\
& \mid if\ S_1\ then\ S_2\ else\ S_3 \\
& \mid while\ S_1\ do\ S_2 \\
& \mid do\ S_1\ while\ S_2 \\
& \mid l :\ S \\
& \mid switch\ S_1\ case\ S_2 \dots\ case\ S_k \\
& \mid try\ S_1\ catch\ S_2 \dots\ catch\ S_k\ finally\ S_{k+1}
\end{aligned}
$$

There is no notation of an expression in the analysis language. Thus, in the condition for an if statement stands the analysis statement generated by the condition expression.

7.2 Flow Graph

The flow graph is a graphical representation of a given analysis statement on which the data flow analysis operates. The nodes of the graph are the elementary analysis statement. The edges represent possible control flow transitions from one node to another. Since in comparison to the existing system we only have introduced two new analysis statements $restoreFields$ and $invRestore(x)$ the creation of the flow graph stays like mentioned in the corresponding section in [12]. Thus, we only note here how to handle the two analysis statements $restoreFields$ and $invRestore(x)$.

Each elementary statement is labeled by an index i. We use the notation $[S]^i$ to denote an elementary statement S with index i. The function $nodes$ yields the indexes of the set of nodes of an analysis statement. Additionally $init$ returns the index of the first node and $final$ the index of the last node. For $restoreFields$ and $invRestore(x)$ these three functions are defined as follows:

$$
\begin{aligned}
nodes([restoreFields]^i) &:= i \\
nodes([invRestore(x)]^i) &:= i \\
init([restoreFields]^i) &:= i \\
init([invRestore(x)]^i) &:= i \\
final([restoreFields]^i) &:= i \\
final([invRestore(x)]^i) &:= i
\end{aligned}
$$

The function $edges$ defines the directed edges between node indexes in the flow graph corresponding to an analysis statement. Since $restoreFields$ and $invRestore(x)$ consists only of one node there are no additional edges between the nodes of these statements. Thus, $edges$ is defined as follows:

$$
\begin{aligned}
edges([restoreFields]^i) &:= \emptyset \\
edges([invRestore(x)]^i) &:= \emptyset
\end{aligned}
$$

7.3 Analysis Variables of Interest

For each method we define a set of variables which we are interested in during the data flow analysis. These so called variables of interest are the following:

$$\mathbf{AVar} = \mathbf{ALoc} \cup \mathbf{AFd} \cup \mathbf{AMark}$$

where:

- **ALoc** - is the set of local variables
- **AFd** - is the set of visible fields, parameters and result of type `rep`
- $\mathbf{AFd}_0$ - is the set of fields of type `rep`
- **AMark** $= \{Cl_{any},\ Cl_{peer},\ unusable,\ nonTransCls(C)\}$ - set of markers

In contrary to the old solution we only have markers for all non-transferable clusters. For all transferable clusters, this means all clusters defined by a `uniq` field, we omit the markers. This allows us to easily distinguish between transferable and non-transferable clusters as only the latter have a marker.

7.4 Partition Sets and Analysis Definition

7.4.1 Set Partitions

One of the possibilities to denote analysis values are partition sets. You will find a deeper introduction to partition sets in the corresponding section in [12]. Each analysis value is represented by a set of partitions. Each partition stays for one possible constellation. The different sets inside a partition are called blocks. If two analysis variables are in one block this means that they point into the same cluster at this program point.

We denote L^x the block of the partition L that contains x. It follows from the partition definition that there is always exactly one such block. With the blocks we can determine the actual type of a variable. This means that if a variable is in the same block as the marker Cl_{peer} it is of type `peer`. If a variable x has no marker in the same block $\mathbf{AMark} \cap L^x = \varnothing$ this means that the variable is of type `rep` $\langle Cl_{tr} \rangle$ for a transferable cluster Cl_{tr}.

Since only non-transferable clusters have a marker, we can never have two markers in the same block:

$$\forall m_1,\ m_2 \in \mathbf{AMark}:\ m_1 \neq m_2 \Rightarrow L^{[m_1]} \neq L^{[m_2]}$$

We check this property explicitly as a postcondition of the merge operation.

There are three operations on partitions: new, merge and move.

Move. The move operation removes a variable x from its block and adds it to the block of y (where A is a set, $L \in Partition(A)$ is a partition and $x, y \in A$ are elements):

$$move(L,x,y) := \begin{cases} L,\ if\ L^x = L^y; \\ L - L^x - L^y + L^x \setminus \{x\} + L^y \cup \{x\},\ otherwise. \end{cases}$$

After the move operation x has the same type as y.

New. The new operation removes a variable x from its block and moves it to a new singleton block:

$$new(L,x) := L - L^x + L^x \setminus \{x\} + \{x\}.$$

After the new operation the type of x changes on `rep` $\langle Cl_{tr} \rangle$.

Merge. The merge operation unifies two blocks x and y:

$$merge(L, x, y) := L - L^x - L^y + L^x \cup L^y.$$

After the merge operation all variables from L^x change to the type of y. We have to ensure that we do not merge two non-transferable clusters. Thus, we have to check the precondition that the two blocks contain no more than one marker:

$$|(L^x \cup L^y) \cap \mathbf{AMark}| \leq 1$$

If the precondition is violated we treat it as a type error.

Additionally we have to guarantee that no `rep` field or parameter changes the type to `peer` or `any`. Thus, if we have a merge where x or y is of type `peer` or `any` all fields of type `rep` in L^x or L^y become unusable:

$$\exists m \in ((L^x \cup L^y) \cap \{Cl_{any}, Cl_{peer}\}) \Rightarrow \forall f \in (L^x \cup L^y) \cap Fd : m_f \neq m \Rightarrow move(L,\ f, unusable)$$

In addition we have three functions for maintaining the object invariant:

InvTest. With the predicate $InvTest$ we test an object's invariant. This means that two fields in the same block have to be declared of the same type and for each field the type inferred by the analysis should be assignable to the declared type:

1. $\forall f_1, f_2 \in Fd_0\ L^{f_1} = L^{f_2} \Rightarrow m_{f_1} = m_{f_2}$
2. $\forall f \in Fd_0\ L^f \leq_A m_f$

We check explicitly that this predicate is true before method invocation on a receiver of type `peer` or `this` as well as before return from a method.

InvRestore(x). The operation $InvRestore(x)$ calls $InvRestore$ if x is `peer`:

1. $isPeer(x)\ != 0 \Rightarrow InvRestore$

InvRestore. With the operation $InvRestore$ we can restore the object's invariant:

1. $\forall v \in Loc\ (\exists f \in Fd_0 : L^v = L^f\ \wedge\ m_f = \texttt{rep}\,\langle Cl_{tr}\rangle) \Rightarrow move(L, v, Cl_{any})$
2. $RestoreFields$

First we change type of a variable to `any` if it belongs to a block which contain a field that refer on a transferable cluster. This avoids having a writable reference into a transferable cluster from the middle of a program stack. Second the operation $RestoreFields$ restores the clusters of the fields.

RestoreFields. With the operation $RestoreFields$ we can restore the clusters of the fields:

1. $\forall f_1, f_2\ m_{f_1} = m_{f_2} \Rightarrow merge(L, f_1, f_2)$

If two fields of same declared type are in different blocks we merge these two blocks. We restore the object's invariant before each method invocation.

7.4.2 Analysis Values

The analysis computes for each node in the flow graph the analysis value that holds at the entry to the node and one that holds at the exit of the node. The analysis value at the exit of a node can be computed by applying the transition function of this node to its analysis value at the entry. The entry analysis value of a node is worked out by joining the exit values of its predecessor nodes.

To distinguish the transferable clusters from the non-transferable clusters all non-transferable clusters contain a marker analysis variable $m \in \mathbf{AMark}$.

7.4.3 Queries

As in the existing system we use the basic predicate $areInSameBlock : 2^{Part(A)} \times A \times A \rightarrow \mathbb{T}$ to check whether two variables $x, y \in \mathbf{PVar}$ are in the same block of partitions contained in a given partition set Φ. The query functions *isUnusable* - which yields whether a variable is unusable - and *pointsInto* - which returns whether a variable point into a given cluster - are used in the same way as in the existing system. We additionally make use of the three functions *isPeer*, *isAny*, and *isRep* to determine the inferred type of a local variable. These functions are defined as follows:

$$isPeer(\Phi, x) := areInSameBlock(\Phi, x, Cl_{peer})$$

$$isAny(\Phi, x) := areInSameBlock(\Phi, x, Cl_{any})$$

$$isRep(\Phi, x) := \begin{cases} 0, & isPeer(\Phi, x) = 1 \vee isAny(\Phi, x) = 1 \vee isUnusable(\Phi, x) = 1 \\ 1, & isPeer(\Phi, x) = 0 \wedge isAny(\Phi, x) = 0 \wedge isUnusable(\Phi, x) = 0 \\ \frac{1}{2}, & else \end{cases}$$

7.4.4 Partition Set Invariants

We have the partition invariant that a block can never contain more than one marker. For a partition set Φ this means:

$$\forall m_1, m_2 \in \mathbf{AMark} : m_1 \neq m_2 \Rightarrow areInSameBlock(\Phi, m_1, m_2) = 0$$

We check the maintaining of this invariant explicitly at the end of the merge operation.

7.4.5 Transition Functions

As described in [12] all transition functions can be expressed using *Merge* and *New* as building blocks. This holds for the newly introduced functions *InvRestore*, *InvRestore*(x), and *RestoreFields* too (where *cluster*(f) yields the declared cluster of f):

$$\begin{aligned} InvRestore(\Phi, x) :=\ & areInSameBlock(\Phi, x, Cl_{peer}) \mathrel{!}= 0 \Rightarrow\ InvRestore(\Phi) \\ InvRestore(\Phi) :=\ & \{\forall v \in \mathbf{ALoc}(\exists f \in \mathbf{AFd_0} : (areInSameBlock(\Phi, v, f) = 0 \wedge \\ & \nexists m \in \mathbf{AMark} : areInSameBlock(\Phi, m, f) = 1)) : Move(\Phi, v, any)\} \\ & \cup RestoreFields(\Phi) \end{aligned}$$

$$RestoreFields(\Phi) := \{\forall f_1 \in \mathbf{AFd_0}, f_2 \in \mathbf{AFd}_0(cluster(f_1) = cluster(f_2)) : Merge(\Phi, f1, f2)\}$$

by remembering that *Move* can be expressed using *Merge* and *New*.

7.4.6 Preconditions

To maintain the partition invariants we have to check the following preconditions before using the transition function $Merge(x, y)$:

$$\forall Cl_1, Cl_2 \in \mathbf{AMark} : pointsInto(\Phi, x, Cl_1) \neq 0 \land pointsInto(\Phi, y, Cl_2) \neq 0 \Rightarrow Cl_1 = Cl_2$$

7.5 Analysis Values

In the existing type system there are different possibilities to represent the analysis values: partition sets, alias matrices and minimized partition sets. Since we change nothing about them, we refer for further reading to the corresponding chapter in [12].

Chapter 8

Implementation

8.1 Existing Universe Type System with Ownership Transfer

In this section we give an introduction how MultiJava and the Universe uniqueness extensions are implemented. Beginning with section 8.2 we show some implementation details about the extension presented in this report.

8.1.1 Packages

The most important package of the MultiJava is the package 'org.multijava.mjc' which contains the files for the MultiJava compiler. The files of this package can be basically divided into four groups [6]:

- The *JPhylum* hierarchy contains all classes representing abstract syntax tree (AST) nodes. Instances of these classes are the output of the parser. All these classes are marked with a 'J' as first letter of the class name. Examples are *JFieldDeclaration*, *JIfStatement* or *JLocalVariable*.
- The *CType* hierarchy contains classes that represent types. For instance is *CArrayType* used for the type of an array or *CClassType* represents a class type.
- The *CContext* hierarchy is used for control flow analysis and variable scoping during type checking.
- The classes of the *CMember* hierarchy represent the signatures of classes, interfaces, fields, and methods. *CField*, *CMethod*, and *CClass* belong to this group of files.

Further we need the classes related to the universe type system with ownership transfer like for the data flow analysis. They can be found in packages beginning with "org.multijava.universes.uniqueness'.

8.1.2 Passes

The MultiJava compiler uses different separate passes to process the AST. The relevant passes for uniqueness type checking are illustrated in figure 8.1. For further reading we refer to [12].

Universe Modifier Translation. During this process - called *InitUniverseUniquenessTask* - the translation from surface Universe modifiers to core modifiers as explained in section 6.3 is done. This task is implemented in the *initUniverseUniqueness* methods of the relevant Java AST classes.

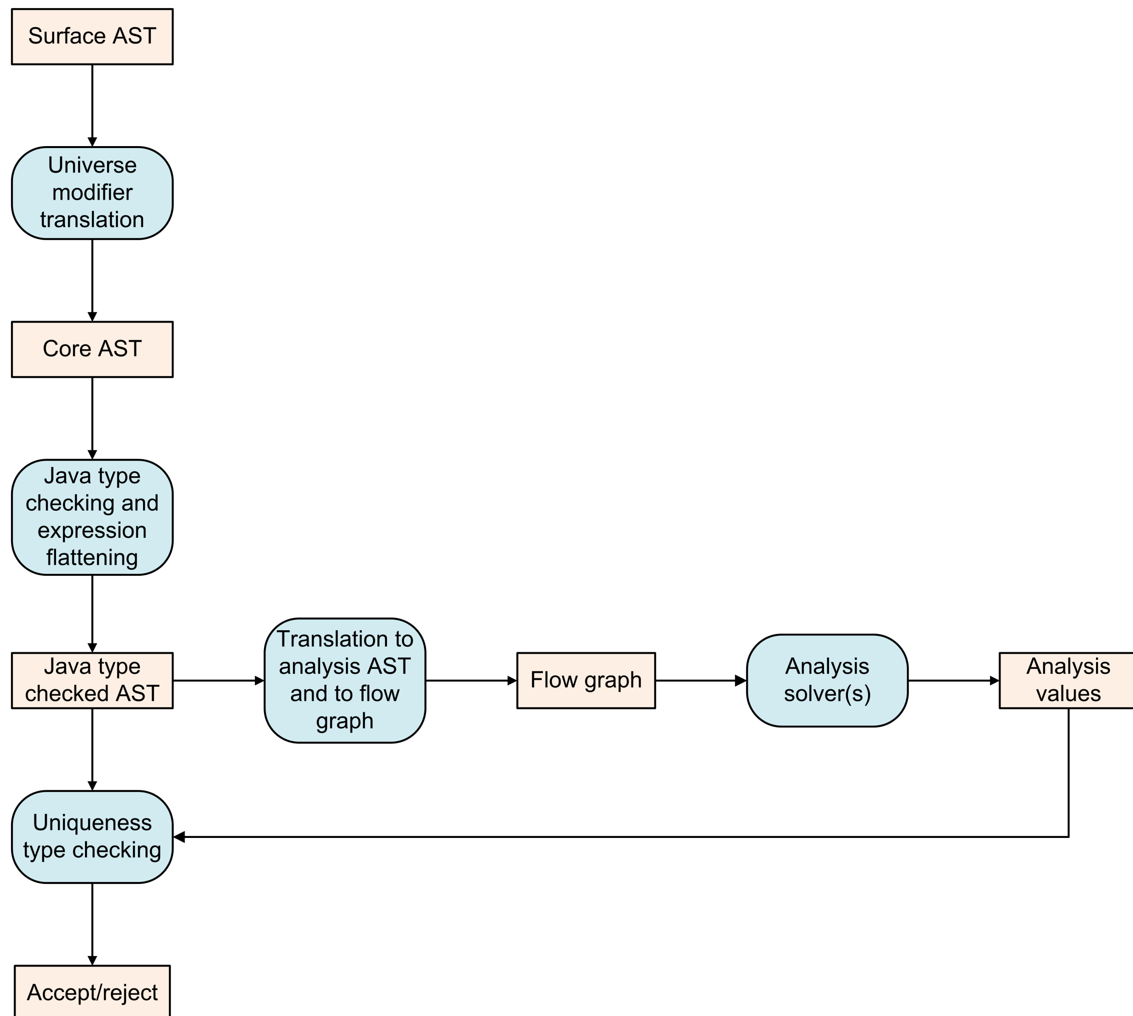

Figure 8.1: Type checking process for the uniqueness type checking. The angled rectangles represent the data while the rounded rectangles are used for operations.

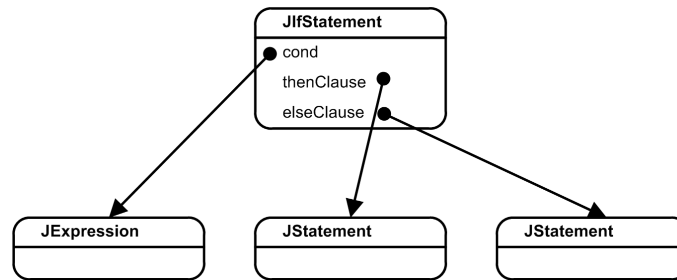

Figure 8.2: The object structure of an if statement in surface and core AST.

Java Type Checking and Expression Flattening. First of all the basic Java type checking is performed. At the same time some type checks for the Universe type system - the basic system without ownership transfer - is done. Both is related to the method *typecheck* of the relevant AST classes.

Additionally the expression flattening is performed. This means that the Java expressions are transformed into statements of toy language by using new temporary variables. The original Java expressions are not modified since the flattening is stored in association with the original expression. This step happens in the method *flatten* of the relevant subclasses of *JExpression*.

Checking Uniqueness The data flow analysis and the remaining uniqueness type checks are performed in a new compiler pass called *CheckUniverseUniquenessTask*. The corresponding functionality is implemented in the method *checkUniverseUniqueness* of the class *JMethodDeclaration*. This pass is divided into three operations. At first the flow graph is created. Then we run the analysis and compute the solution of the data flow analysis. At the end the uniqueness type checking is performed using the computed analysis values. It is performed on the flattened AST, not on the original Java AST.

Translation to Analysis AST and to Flow Graph. For each method declaration, its body is translated into corresponding statements of the analysis language according to the transition rules. This happens in the methods *getAnalysisStmt* and *createAnalysisStmtNoFlattening* of the corresponding AST classes.

Uniqueness Type Checking. If the analysis could successfully be solved, the uniqueness type checking according to the type checking rules is performed using the analysis values computed by the solver. This type checking is implemented in the method *checkUniverseUniquenessNoFlattening* of the corresponding AST classes.

8.1.3 Object Structure

In this section we want to show how the data between the different operations shown in figure 8.1 are represented. We clearly include only the most important objects and references.

Surface and Core AST The surface AST is the output of the parser. The difference of surface and core AST are only little changes according to the universe modifiers. Thus, the whole object structure stays quite the same. In figure 8.2 you find the object structure of an `if` statement. In figure 8.3 the object structure of a field declaration is illustrated. On the one hand you see that the universe types are linked by the field *universe* of the class *CClassType* which represents a Java class type. On the other hand each local variable has a reference to an object of type *LocalAnanlysisVar* which is used in the data flow analysis to represent this variable.

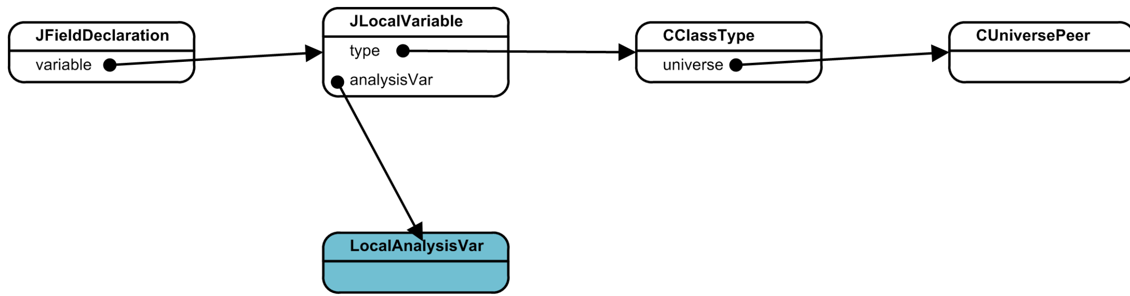

Figure 8.3: The object structure of a field declaration in surface and core AST.

Analysis AST The analysis AST is the representation of the program in terms of analysis statements which is used to create the flow graph. Each statement and expression of the core AST holds a reference to its analysis statement. The analysis language consists of the analysis statements introduced in section 7.1 and the analysis variables explained in section 7.3. A possible object structure of the analysis AST is presented in figure 8.4. We differ between the analysis statement which are marked yellow and green (yellow are the elementary analysis statements and green the composite analysis statements) and the blue analysis variables. You can see that the objects of type $MergeStmt$ and $MoveStmt$ have references to their analysis variables. These analysis variable objects are the same as the $LocalAnalysisVar$ object shown in figure 8.3. Each analysis node has three references $initNode$, $innerNode$, and $finalNode$ as mentioned in section 7.1. Notice that these references point only to elementary analysis statements which can be seen like the leafs of the analysis statement tree and are marked yellow in our example. In figure 8.3 we show only these three references for the top sequence statement to not worsen the understandability.

FlowGraph The flow graph connects the elementary analysis statements from the analysis AST according to the program control flow. Since the elementary statements implement the interface $FlowGraphNode$ all elementary statement objects are of type $FlowGraphNode$ too. Thus, the terms flow graph node and elementary analysis statement can be used as synonyms.

An example for a flow graph is shown in figure 8.5. Each flow graph node has a reference to all its outgoing edged. Objects of type $FlowGraphEdge$ represent an directed connection between two flow graph nodes.

If you compare the figures 8.5 and 8.4 you remark that for creating the flow graph all elementary analysis statement objects - marked as yellow - are taken and connected with each other through flow graph edges. Thus, we have not one analysis AST and one flow graph for each method, but both are strongly overlapped.

How the object of type $FlowGraph$ which represents the flow graph of one method is represented is illustrated in figure 8.6. We leave away the connection between the objects referred by the flow graph object since we have shown them already in the last figures. For instance the flow graph edges and the flow graph nodes are connected. As mentioned above the object of type $AnalysisStmt$ has references to the elementary analysis statements represented by $FlowGraphNode$ objects. These nodes have additional links to the different analysis variables of type $AnalysisVar$.

Solver After the creation of the flow graph we can run the analysis. The result of the analysis is the analysis values. Each analysis value represents the state of the clusters and analysis variables at one point of the program. The existing system can use three different solvers. In the figure 8.7 the object structure of the alias matrix solver is shown. You see in the middle of the figure the array of $AliasMatrix$ objects. These are the analysis values that hold which variables point into which cluster. Due to conditional constructs more than one cluster state is possible at one program point. These different possible states are stored in the array referred by the field

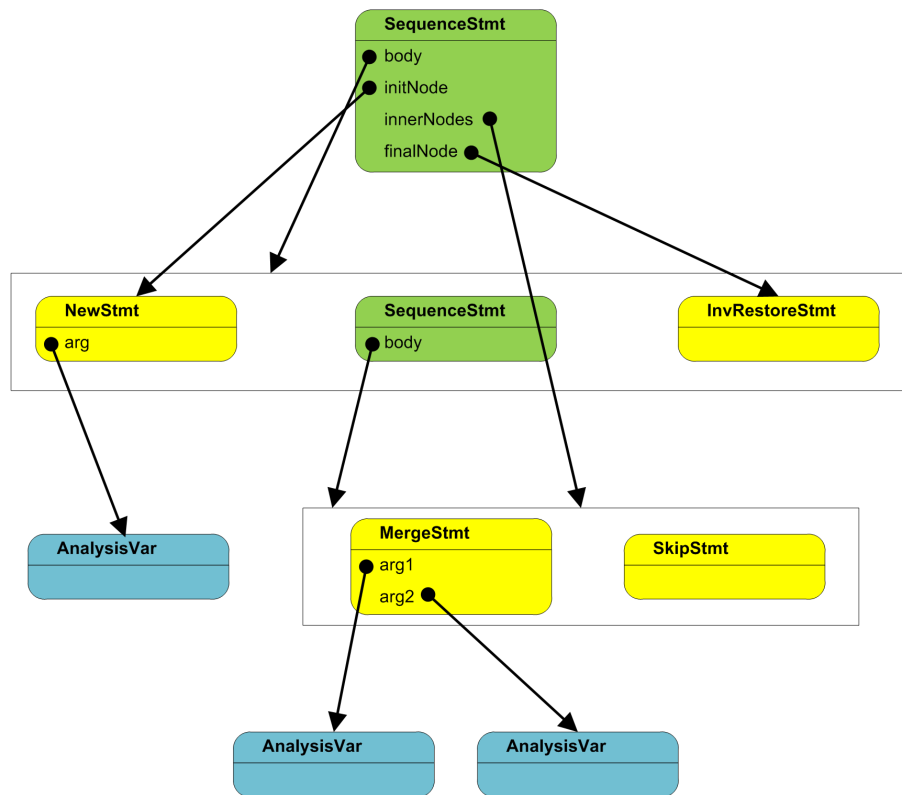

Figure 8.4: An object structure of the analysis AST. The rectangles represent arrays.

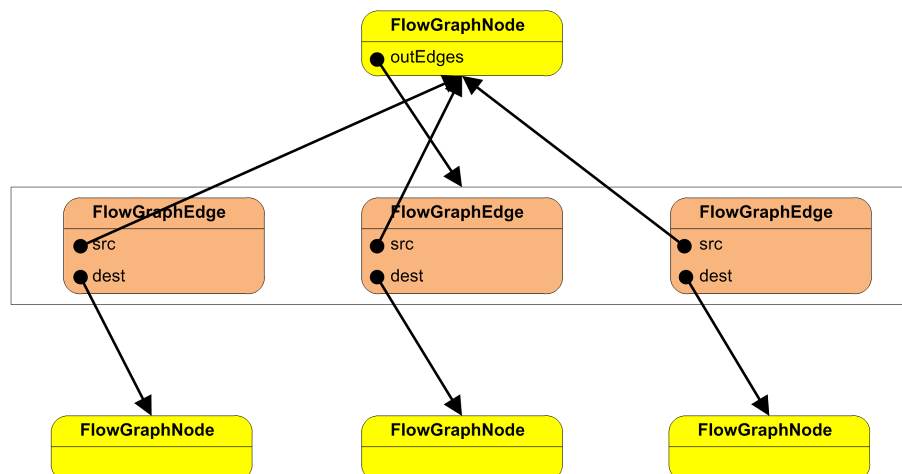

Figure 8.5: A part of the flow graph. The flow graph nodes are the same objects as the elementary analysis statements in the analysis AST in figure 8.4.

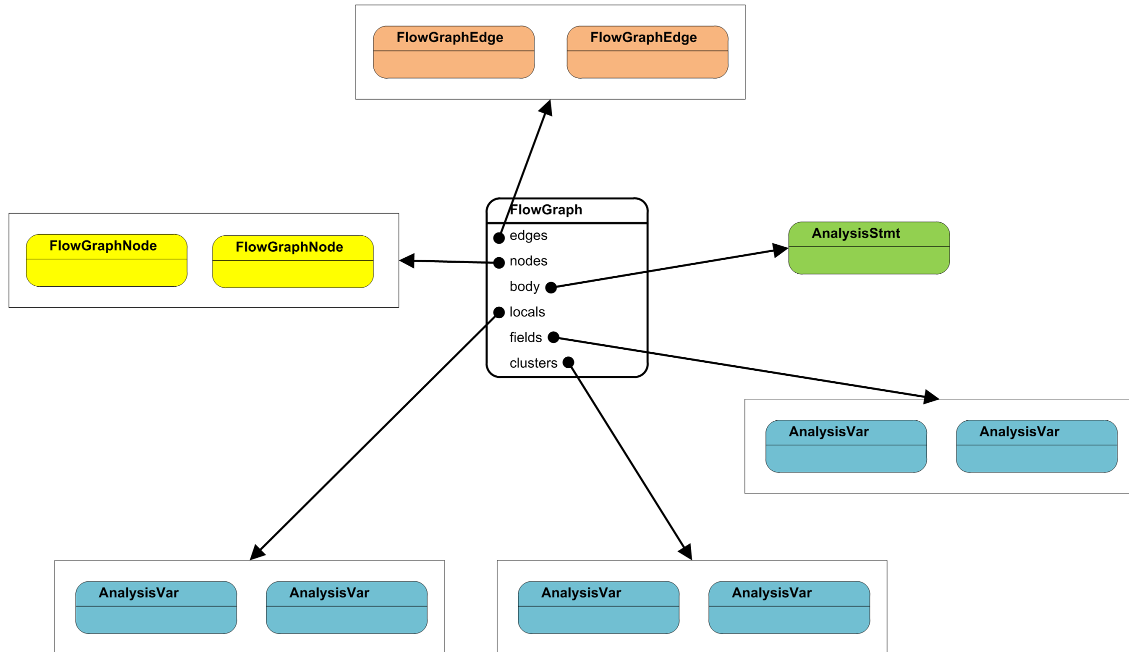

Figure 8.6: Object structure of one flow graph.

aliasSet. The access to the analysis values during uniqueness type checking occurs over the object *CUniverseUniqContext*.

8.2 General

Our implementation extensions affect only the MultiJava code since JML automatically benefits from them. We used Java version 5 (generics and autoboxing) while the existing files were written for Java 1.4.

8.3 Local Variable Inference

8.3.1 Universe Modifier Translation

Modifiers before a local variable during the declaration or in combination with the `new` operator are not needed anymore. For backward compatibility they should be accepted, but ignored. The user will be warned that the modifier is omitted. We implement this in the method *typecheck* of the class JVariableDeclarationStatement corresponding to the way the correct use of modifiers before local variables is checked in the old version. In the same way we can check for `new` operation in the method *typecheck* in JNewObjectExpression if there is a modifier for local variable object creation.

All variables are linked with a modifier object at time of the surface AST creation, either with an explicit (and now ignored) modifier or with the implicit default modifier. We do not delete this modifier since for future work it can probably be used as a type hint, but we will now ignore it.

8.3.2 Java Type Checking and Expression Flattening

Since during the task of the Java type checking and the expression flattening the Universe modifiers of the local variables are not inferred yet, it is important that all Universe type checking stuff

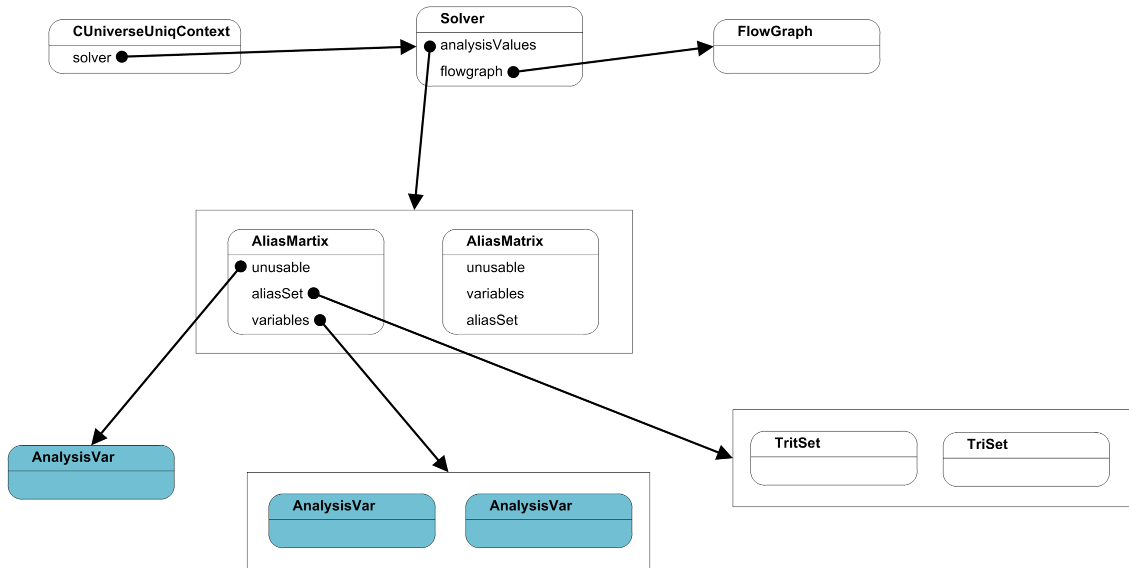

Figure 8.7: Object structure of a solver using alias matrix as analysis value.

is moved to the Uniqueness type checking pass (this means a move from typecheck() to checkUniverseUniqueness()). While *typecheck* runs on the original tree, *checkUniverseUniqueness* uses the flattened version. But is works fine too. The movement of the code has the consequence that the type checking is done after the analysis and thus the errors and warnings have another order that before. Additionally we have to change the type queries for local variables as well as `rep` fields and input parameters to queries to the analysis data flow.

isLValue. In the method *typecheck(CExpressionContextType context)* there is a call to the method *isLValue(CExpressionContextType context)*:

```
1       public JExpression typecheck(CExpressionContextType context) {
2               ...
3               boolean isLValue = isLValue(context);
4               ...
5       }
6
7       public boolean isLValue(CExpressionContextType context) {
8               ...
9               if (context.isPure()) { ... }
10              ...
11      }
```

Since *isLValue* needs type information we have to move this method call too. The *isLValue* method needs as argument an object of type *CExpressionContextType* for example to check whether we are in a pure context as you can see in line 9. In the method *checkUniverseUniqueness* we miss the context object. Thus, we cannot call directly the *isLValue* method. We implemented it in that way that in the method *typecheck* we call a new method *setLValueFlags* which stores in private boolean flags the information needed from the *CExpressionContextType* object in the method *isLValue* as for instance *isPure*. We now call from *checkUniverseUniqueness* a new method *isLValue* which has no argument and takes the information required from the *CExpressionContextType* object by reading out the flags. After these changes the example would look like this:

```
public JExpression typecheck(CExpressionContextType context) {
        ...
        setLValueFlags(context);
        ...
}

public boolean isLValue() {
        ...
        if (isPure) { ... }
        ...
}

public void setLValueFlags(CExpressionContextType context) {
        ...
        isPure = context.isPure();
        ...
}

protected void checkUniverseUniquenessNoFlattening(
                CUniverseUniqContextType context) {
        ...
        boolean isLValue = isLValue();
        ...
}

private boolean isPure;
```

8.3.3 Analysis Variables of Interest

In the old system only `rep` local variables and `rep` fields are used as analysis variables of interest. We expand this now to all local variables and parameters of type `rep`. Thus we have to register all local variables as well as fields and parameters of type `rep`. In the flow graph we handle parameters like fields. The initial value of the partitions is that each field and parameter is in its cluster set and each local variable is in an own set. The only exception occurs with multiple references on one `free` cluster. All these references point into the same cluster at the beginning.

8.3.4 Translation to analysis AST

The analysis transition functions used to translate the core AST to the analysis AST has to be modified since we have now no knowledge about the modifiers of the local variables (–> createAnalysisStatement() of subclasses of JStatement). The modified rules are listed in section 6.9. The different type rules are implemented in the method *createAnalysisStmtNoFlattening* of the classes mentioned in figure 8.1.

In the analysis transition rules (see section 6.9) we use the type combinator in a parameter for the *move* and *merge* operations several times which looks as follows: $merge(Pt, x, y \rhd_U m_f)$. In every case the left hand side argument of the type combinator is a local variable and the right hand side is a field, a method parameter or the result variable. Thus, we know the type of the right hand side argument except in which cluster a `rep` variable points. Otherwise we do not know anything about the type of the local variable. We can only check if the local variable is `this` or not. Although we can determine which type the combination of the two variables should have in terms of real type or in term of the type that one of the two involved variables will have at analysis time. This is shown in the following table (the evaluation order must be from top to bottom):

$$\texttt{this} \rhd_U f = f$$

L-ASSIGN:	JLocalVariableExpression
L-FD-RD:	JClassFieldExpression
L-FD-WR:	JClassFieldExpression
L-ARR-RD:	JArrayAccessExpression
L-ARR-WR:	JArrayAccessExpression
L-PRE-INVK:	JMethodCallExpression, JExplicitConstructorInvocation
L-INV:	JMethodCallExpression
L-NEW:	JNewObjectExpression
L-NEW-ARR:	JNewArrayExpression
L-NULL:	JUnaryPromote
L-CAST:	JCastExpression
L-RETURN:	JReturnStatement

Table 8.1: Classes where the transition rules are implemented.

$$x \rhd_U \texttt{any} = \texttt{any}$$
$$x \rhd_U \texttt{free} = \texttt{free}$$
$$x \rhd_U \texttt{rep} = \texttt{any}$$
$$x \rhd_U \texttt{peer} = x$$

The analysis variables for the local variable and the field can be detected at the time when the transition from surface AST to analysis AST takes place. But since we do not have already the analysis variables of the two markers Cl_{peer} and Cl_{any} we have to introduce objects of type $AnalysisVarPeer$ and $AnalysisVarAny$ as placeholder for the two markers. At analysis time this objects forward queries to the marker objects of the $AnalysisValue$ at this point of the computation.

The *merge* operation should be extended. All clusters which are not transferable are marked with a marker (Cl_{peer}, Cl_{any}, *unusable* or the `this` clusters of a class). To make sure that not two non-transferable clusters are merged we check as postcondition of *merge* whether in the merged set there is more than one marker. This would mean a type error.

After the *merge* operation we have to make sure that all `rep` fields whose types changed to not `rep` should be made unusable. This is done by the method $filterUnusableFields$ which is invoked by the *merge* operation.

8.3.5 Analysis Solver and Partition Sets

The analysis solver works on the analysis AST and the partition sets. We only need to handle the two additional markers Cl_{peer} and Cl_{any}.

Each partition set have to be extended with two new sets marked as holding all local variables which are `peer` or `any`.

8.3.6 Performing the Type Checks

As mentioned above the type checking rules in the method $checkUniverseUniquenessNoFlattening$ have to be adapted as shown in figure 8.2.

Since we have no stable type information about local variables, we need to take the information from the analyzer. By checking if a local variable is in the cluster Cl_{peer}, in the cluster Cl_{any} or in one of the `this` clusters we can find out if it is `peer`, `any`, `rep` $\langle Cl_{nonTr} \rangle$ or `rep` $\langle Cl_{tr} \rangle$ at this point.

8.3.7 Parameter

In the old version parameters were handled like variables. Each parameter is represented by an object of type JFormalParameter which is a subclass of JLocalVariable. Expressions containing

T-ASSIGN:	JLocalVariableExpression
T-FD-RD:	JClassFieldExpression
T-FD-WR:	JClassFieldExpression
T-ARR-RD:	JArrayAccessExpression
T-ARR-WR:	JArrayAccessExpression
T-INV:	JMethodCallExpression, JExplicitConstructorInvocation
T-NEW:	JNewObjectExpression
T-NEW-ARR:	JNewArrayExpression
T-NULL:	JUnaryPromote
T-CAST:	JCastExpression
T-RETURN:	JReturnStatement
T-MDECL:	JMethodDeclaration, JReturnStatement

Table 8.2: Classes where the type rules are implemented.

a parameter were described by an object of type JLocalVariableExpression like it was for local variables. This object structure is shown by the light gray ellipses in figure 8.8.

We now want to handle parameters for the Universe uniqueness stuff like field access. Thus, for the normal Java checks we handle formal parameter like local variable, while the creation of the analysis statements and the Universe checking is done like for fields. Therefore, we represent parameters by objects of type JParameterExpression - a subclass of JClassFieldExpression - in the flattened version. The additionally for the flattening created object are marked dark-gray in the figure 8.8.

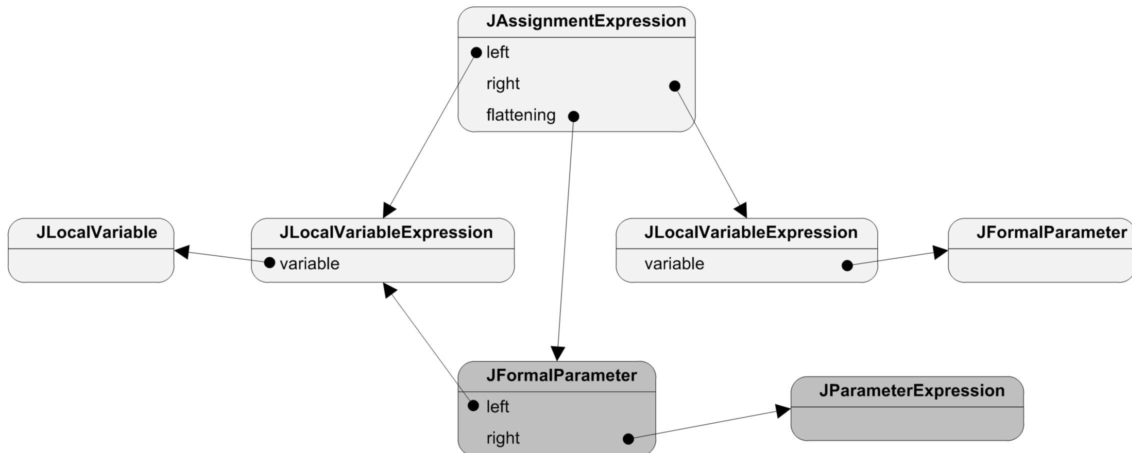

Figure 8.8: Slightly simplified object structure for parameters. For the Java checking part the light gray structure is used where parameters are handled like local variables. The Universe checking is executed on the flattened version where parameter access is treated like field access. The additional objects for the flattened version are marked dark-gray.

8.3.8 Multiple Universe Types

In the old version the Universe type of a variable is represented by an instance of type *CUniverseRep*, *CUniversePeer*, and *CUniverseReadonly*. This works fine as long as each variable has one clear type. Since queries to the analysis can return *DONT_KNOW* it is possible that an object has more than one possible type. For these cases we create a new class *CUniverseMultipleTypes* that represent a consolidation of different Universe types. As *CUniverseMultipleTypes* implements - like *CUniverseRep*, *CUniversePeer*, and *CUniverseReadonly* - the abstract meth-

ods of $CUniverse$ it can be handled like a normal Universe type. The only difference is that $CUniverseMultipleTypes$ has additional methods like $isDefinitelyRep$. The difference between $isDefinitelyRep$ and $isRep$ is that $isRep$ return true if the variable may be `rep` regardless whether it is probably of another type too while $isDefinitelyRep$ returns true iff the variable is only of the type `rep`. The class structure is shown in figure 8.9.

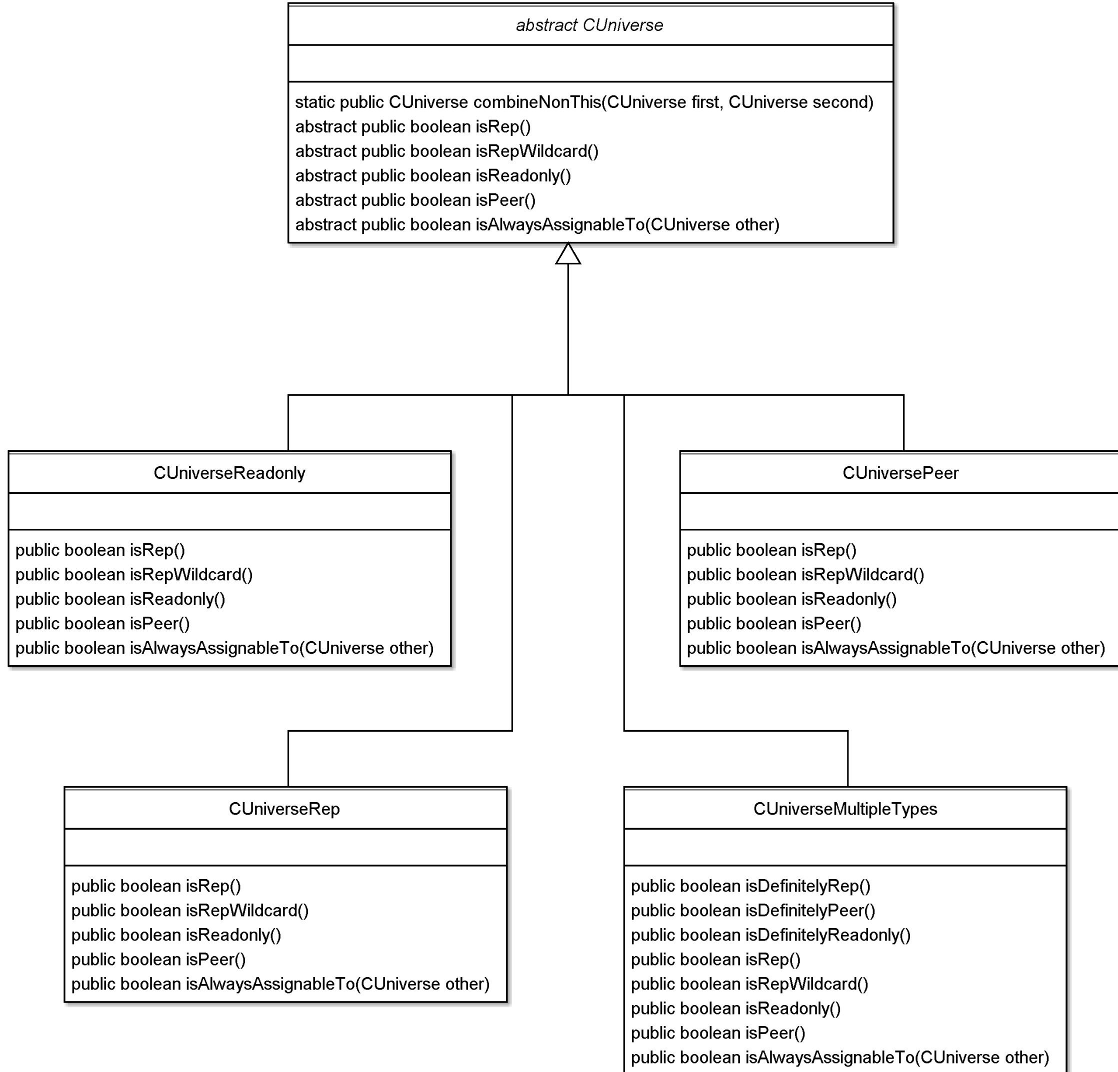

Figure 8.9: Class diagramm of the Universe type classes.

8.3.9 InvTest

The invariant test $InvTest$ includes the following two rules:

1. $\forall f_1, f_2 \in Fd_0 \; L^{f_1} = L^{f_2} \Rightarrow m_{f_1} = m_{f_2}$
2. $\forall f \in Fd_0 \; L^f \leq_A m_f$

We implement this test in the class $AbstractAnalysisValue$. The straightforward solution would be to do both checks using the $CUniverse$ object the field was declared with. But in the class $AbstractAnalysisValue$ we have only access to the analysis variables. Thus, we have only field

analysis variables for each `rep` field. In addition we know for each field analysis variable the cluster which it was declared for. We have to use this cluster for looking up the declared type.

While the comparison, if two field analysis variables are declared of the same type, is easily done by the check of its declared clusters, the assignable-to relation in the second check needs some additional thought. Because `peer` and `any` fields can never change their type we can leave out and only care about `rep` fields which are covered by the field analysis variables. If such a field is in a transferable cluster it would be always assignable to its declared type since `rep` $\langle Cl_{tr} \rangle$ is assignable to every type. Otherwise if a field is in a non-transferable cluster it should always be declared of this cluster. This means that for the assignability check we have only to look that a field declared as pointing into a non-transferable cluster is in its declared cluster now. In addition no field should be unusable. The whole code for method *invText* is listed in figure 8.10.

```
public void invTest(TokenReference sourceRef){
    for (int i = 0; i < nofFields; i++) {
        FieldAnalysisVar field1 = getField(i);
        for (int j = i + 1; j < nofFields; j++) {
            FieldAnalysisVar field2 = getField(j);
            // P_f1 = P_f2 > m_f1 = m_f2
            if (pointToSameCluster(field1, field2) != NO
            && !field1.getCluster().equals(field2.getCluster())) {
                // report error
            }
        }

        // type inferred by analysis should be assignable to declared type
        checkNotUnusable(field1, sourceRef);

        if (pointsPossiblyIntoNonTransferableCluster(field1)
        && pointToSameCluster(field1,
            field1.getCluster().getAnalysisVar()) != YES) {
            // report error
        }
    }
}
```

Figure 8.10: Code of invariant test method in class AbstractAnalysisValue.

The implementation of the methods *invRestore* and *restoreFields* follows the same principle as for *invTest* and take the declared cluster of an analysis variable to get the declared Universe type of the variable.

8.4 Subclass Separation

Each class of type *CClass* is at creation time linked with a own `this` cluster. During the *initUniverseUniqueness* pass each field, parameter and result type declared as plain `rep` is linked with the `this` cluster of its owner class.

For the analysis of one method we linked all `this` clusters during execution of method *registeringRepClusters-AndRepFieldsDone*() with a cluster analysis variable.

8.5 Arrays

8.5.1 Array Type Object

Each array variable is linked with an object of type *CArrayType* which represent its type. As shown in figure 8.11 an array type object has three important fields *baseType*, *result_type*, and *universe*. *baseType* represents the declared type of the elements, while *universe* is the ownership type of the array object. *result_type* is the type needed to access the elements. This means it is the combined type of the array object type and the element type.

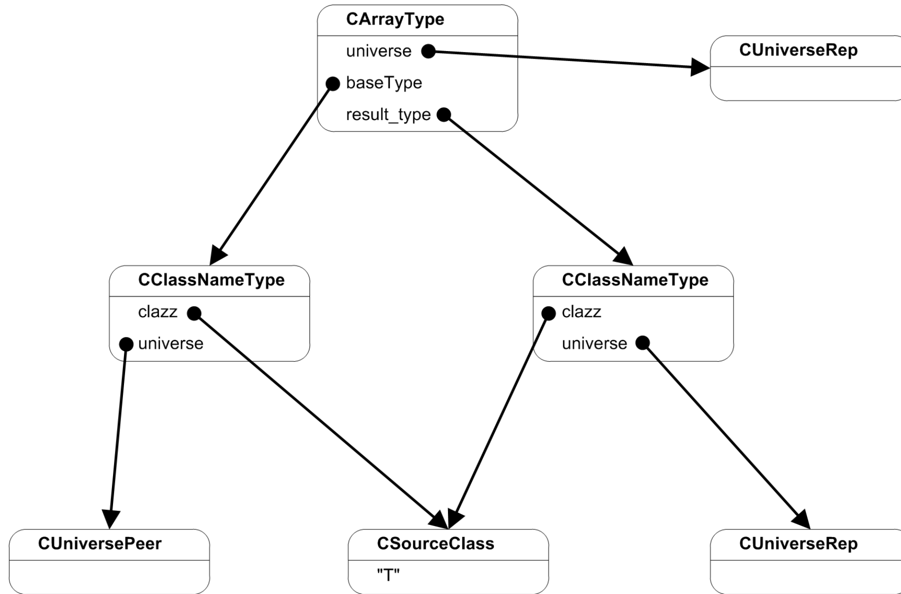

Figure 8.11: Structure of the type object for the array rep peer T[].

In the class *CArrayType* we had to change the assignability relation according to figure 4.2.

8.5.2 Array Access

The two transition rules for array writing and array reading are implemented in the method *createAnalysisStmtNoFlattening* of the of the class *JArrayAccessExpression*. Analogously the type rules are added to the method *checkUniverseUniquenessNoFlattening* of the same class.

8.5.3 Array Creation Expression

In the class *JNewArrayExpression* on the one hand we have to implement the flattening process in the method *flatten*. On the other hand we have to implement the transition rule in the method *createAnalysisStmtNoFlattening* and the type rule in the method *checkUniverseUniqueness-NoFlattening*.

8.5.4 Array Initializer

For array initialization it is only needed to implement the method *flatten* in the class *JArrayInitializer*. Since in the flattened tree there are no nodes of type *JArrayInitializer* no type rules or transition rules are needed to be implemented in this class.

8.6 Passing Multiple References on a Free Cluster

8.6.1 Type Declaration Control

During parsing there is no difference whether a parameter p declared as `rep`[x] means that it points into the same cluster as a `free` parameter x or as a `uniq` field x. During the Universe uniqueness initialization pass in the method *initUniverseUniqueness* of the class *JFormalParameter* we check whether there is a parameter called x in the same method declared as `free`. If there is no such parameter we continue like in the old version by searching for a field x declared as `uniq` in the same class. But if we find such a parameter x we first change the type of p to `free` $\langle x \rangle$. Then we store a reference to parameter p in the object x. This link is later needed for efficiently finding all parameters that point into the same `free` cluster.

8.6.2 Initial Situation

We have to ensure that at the beginning of a method all parameters that point into the same cluster are in one block. Thus in the method *checkUniverseUniqueness* of class *JMethodDeclaration* we create the analysis statements to merge all parameters that point into the same `free` cluster. These analysis statements are added at the beginning of the statement sequence that is created by the body of the method.

8.6.3 Parameter Order

As mentioned in section 5.2.3 we have to change the order of the analysis statements created by each parameter passing in such a way that all formal parameters pointing into the same `free` cluster are handled successively. In addition the analysis statement before which the assignability check has to be done should be for each parameter of such a group the first analysis statement of the group.

Order of Analysis Statements. The ordering of the analysis statements is done in the method *createAnalysisStmtNoFlattening* of the class *JMethodCallExpression*. During the creation of the merge statements for each parameter we arrange them into three different lists. In one list we hold all analysis statements of parameters declared plain `free`, in another one all statements of parameters of type `free` $\langle p \rangle$. In the main list we put the statements of all other parameters. Then the analysis statement of each `free` parameter is added to the main list followed by all parameters pointing into the same cluster. To efficiently finding the parameters pointing into the same `free` cluster we use the method *getMultipleFreeParameters* in *JFormalParameter* which gives back all these formal parameters. To find the analysis statements connected with each of these parameters we use an array *stmtForParameters* that maps the original parameter indexes to the previously created statements. This means that for a method call:

```
void m(free T f, peer T p, rep[f] T f2, free T g, rep[f] T f3, rep T r, rep[g] T g2)
```

The order of its parameters according to the analysis statements would be:

```
(peer T p, rep T r, free T f, rep[f] T f2, rep[f] T f3, free T g, rep[g] T g2)
```

In the array *checkArgsNodes* we store for each parameter before which analysis statement its type check should happens. For indexing the original parameter order is used. For each parameter which is not of type `free` $\langle p \rangle$ the item in *checkArgsNodes* points to its own analysis statement. The *checkArgsNodes* item for all parameters declared as pointing into the same cluster as another `free` parameter p are set to the analysis statement of p.

Checking Assignability. When we now want to check the assignability in the method *checkUniverseUniquenessNoFlattening* of *JMethodCallExpression* we do this for each parameter before the analysis statement stored in *checkArgsNodes*.

8.7 Changes or New Files

The following files have been changed or added:

- Package org.multijava.universes.uniqueness.analysis (classes for analysis variables and analysis values): AbstractAnalysisValue, AnalysisValue, AnalysisVar, AnalysisVarAny, AnalysisVarPeer, ClusterAnalysisVar, FieldAnalysisVar
- Package org.multijava.universes.uniqueness.analysis.statements (classes for analysis statements): AbstractAnalysisStmtVisitor, AnalysisStmtVisitor, InvRestoreStmt, RestoreFieldsStmt
- Package org.multijava.universes.uniqueness.analysis.solvers.singlepartition (class for the solver): SinglePartitionAnalysisValue
- Package org.multijava.universes.uniqueness.analysis.solvers.partitionset (class for the solver): AbstractPartitionSet
- Package org.multijava.universes.uniqueness.analysis.solvers.aliasmatrix (class for the solver): AliasMatrix
- Package org.multijava.universes.uniqueness.analysis.graph (class for the flow graph): FlowGraph
- Package org.multijava.universes.uniqueness.testing.analysis.solvers (class for testing): TestWorklistSolver
- Package org.multijava.mjc:
  - Classes for Universe types: CUniverse, CUniverseImplicitPeer, CUniverseImplicitReadonly, CUniverseMultipleTypes, CUniversePeer, CUniverseReadonly, CUniverseRep, CUniverseRepCluster
  - AST classes: JArrayAccessExpression, JArrayInitializer, JAssignmentExpression, JCastExpression, JClassDeclaration, JClassFieldExpression, JExplicitConstructorInvocation, JFieldDeclaration, JFormalParameter, JLocalVariable, JLocalVariableExpression, JMethodCallExpression, JMethodDeclaration, JNewArrayExpression, JNewObjectExpression, JParameterExpression, JParenthesedExpression, JReturnStatement, JThisExpression, JTypeNameExpression, JUnaryPromote, JVariableDeclarationStatement, JVariableDefinition
  - Classes and interfaces for types, methods, and classes: CArrayType, CClass, CClassType, CInitializable, CMethod
  - Other classes needed for uniqueness: CUniverseUniqContext, CUniverseUniqContextType, CUniverseUniqMessages, CUniverseUniqUtils

Chapter 9

Conclusion and Future Work

9.1 Examples

9.1.1 Enhanced Linked List

Figure 1.3 shows the implementation of a linked list with the possibility to merge two linked lists. Due to our extension we can enhance this linked list. First of all in our implementation (A.1) we can simplify the releasing of a linked list as explained in section 2.2.11.

We added a second method

concatenate(`free` Node first, `rep`[first] Node last)

which adds not a whole list given as parameter but only a part given by the first and the last list element. This shows the benefit of passing multiple references to one `free` cluster via method invocation.

As a further additional feature we implement a method

`pure` `free` LinkedList find(`free` Iterator it , `any` Element obj)

that searches an element given a list iterator. Inside of this `pure` method we have to call the non-pure method *getNext* on the iterator. Thus, we make use of our enhanced purity.

9.1.2 Hash Table Merging

The code in A.2 shows the merging of two hash tables. In the two *merge* methods we make use of the possibility to transfer whole arrays. First in method call

merge(otherTable.getHashTable());

we release one hash table represented by an array and in the method *merge*:

`void` merge(`free` `peer` LinkedList[] otherTable)

we capture this hash table. Thus we changed the owner of a whole array.

9.1.3 AVL Tree

In the AVL Tree example (A.3) each subtree is owned by its parent node. Thus for each change of the parent node an ownership transfer is needed. Note that the left and the right tree of a node are in different clusters since they are moved independently.

The two methods

`free` AVLTree getLeft() and `free` AVLTree getRight()

are used to release the left or the right tree of a node, while the corresponding methods

`void` setLeft(`free` AVLTree left) and `void` setRight(`free` AVLTree right)

capture a `free` tree. The rotation methods like

`free` AVLTree rotateWithLeftChild(`free` AVLTree k2)

capture a `free` tree, make the desired rotation on it and return the released tree. Similarly works the insert method

`free` AVLTree insert(`int` x, `free` AVLTree t).

It captures a tree, inserts the new node by modifying and rotating the tree and returns the released tree.

9.2 Conclusion

In this project we have completed the existing Universe type system with ownership transfer with different extensions. The biggest extension was the type inference for local variables. Due to it we get a more flexible solution since local variables can change their type and never become unusable. Otherwise we could eliminate minor limitations of the existing system like the problem with the release method in the list example (see section 2.2.11). Another advantage is that the user has less notation overhead since no Universe modifiers for local variables are needed. On the other hand our solution is more complicated to understand for the user. The different handling of local variables and fields can be confusing and thus is another disadvantage of the local variable type inference.

The enhanced purity allows more expressive examples like the $find$ method in the linked list (A.1). In fact, the precondition for the enhanced purity is complex but since it is only an extension to the old purity there is no must for the user to make profit of it. Thus, the benefits of our enhanced purity are bigger.

With the subclass separation we separated the `this` cluster of a class from the `this` clusters of its subclasses. For the user the assignability of variables in the `this` cluster is more restricted. But this feature has the advantage that the invariant of a subclass can be checked without touching the superclass. This would be nice if once the Universe type system is integrated in Spec#.

For array handling we achieved a good solution which is derived from the handling of fields. Array creation and initialization expressions are supported too. There is only the restriction that all array elements should always be in the same cluster.

Our last extension was the support for passing multiple references to one `free` cluster via method invocation. This is a powerful feature which allows methods like *concatenate* in the linked list example (A.1). On the one hand we tried to use the existing syntax for this feature. On the other hand this can be confusing since `rep`[f] does not always refer to a cluster declared by a field.

9.3 Future Work

There are still some extension to the project which are left as future work:

Runtime Checks. In the method *typeCheck* some Universe runtime checks are done. They make use of the declared Universe types. For local variables the type determined in the analysis should be used. Thus, these checks should be moved to the method $checkUniverseUniqueness$-$NoFlattening$.

Exception Handling Model. In the current system exceptions are handled in a primitive way. For the future a more precise treatment is desirable. A possible solution is to transfer an exception into the context of the handler during propagation.

Acknowledgment. The author would like to thank Arsenii Rudich and Prof. Peter Müller for their support.

Bibliography

[1] J. Boyland. Alias burying: Unique variables without destructive reads. *Software - Practice and Experience*, 31(6):533–553, May 2001.

[2] D. Clarke and T. Wrigstad. External uniqueness. 2002.

[3] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 130–145, New York, NY, USA, 2000. ACM Press.

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

[5] W. Dietl, P. Müller, and D. Schregenberger. Universe Type System - Quick-Reference. August 2005.

[6] MultiJava Documentation.

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

[8] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification.* Addison-Wesley Professional, third edition, 2005.

[9] http://research.microsoft.com/specsharp/.

[10] G. T. Leavens, A. L. Baker, and C. Ruby. JML: A Notation for Detailed Design. *Behavioral Specifications for Businesses and Systems*, pages 175–188, 1999.

[11] P. Müller and A. Rudich. Ownership Transfer in Universe Types. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2007. To appear.

[12] Y. Takano. Implementing Uniqueness and Ownership Transfer in the Universe Type System, 2007.

Appendix A

Code Examples

In this three code examples the keyword `any` is replaced by the keyword `readonly` since in the current implementation `readonly` is needed.

A.1 Enhanced Linked List

```
1  import java.util.ArrayList;
2
3  /**
4   * Testing uniqueness, ownership transfer and local variable type inference in the Universe type
5   * system: linked list implementation.
6   * Testing the enhanced purity: method find
7   * Testing multiple references on one free cluster:
8   * method concatenate(free Node first, rep[first] Node last)
9   * <p>
10  * Expected result: no errors
11  *
12  * @author ytakano, schaada
13  */
14
15 public class LinkedList {
16
17     protected uniq Node header;
18
19     pure public LinkedList() {
20         Node tmp = new Node(null, null, null);
21         tmp.next = tmp;
22         tmp.prev = tmp;
23         header = tmp;
24     }
25
26     public void addFirst(readonly Element o) {
27         addBefore(o, header.next);
28     }
29
30     public void addLast(readonly Element o) {
31         addBefore(o, header);
32     }
33
```

```
34      protected void addBefore(readonly Element o, rep[header] Node node) {
35          Node newNode = new Node(o, node, node.prev);
36          newNode.prev.next = newNode;
37          newNode.next.prev = newNode;
38      }
39
40      public boolean isEmpty() {
41          return header.next == header;
42      }
43
44      protected free Node getHeader() {
45          Node result = header;
46
47          header = new Node(null, null, null);
48          header.next = header.prev = header;
49
50          return result;
51      }
52
53      public void concatenate(peer LinkedList other) {
54          if (other.isEmpty()) {
55              return;
56          }
57
58          Node otherHeader = other.getHeader();
59
60          header.prev.next = otherHeader.next;
61          otherHeader.prev.next = header;
62
63          otherHeader.next.prev = header.prev;
64          header.prev = otherHeader.prev;
65      }
66
67      public void concatenate(free Node first, rep[first] Node last) {
68          if ( first == null || last == null) {
69              return;
70          }
71
72          header.prev.next = first ;
73          last .next = header;
74
75          first .prev = header.prev;
76          header.prev = last;
77      }
78
79      public pure free LinkedList find(free Iterator it, readonly Element obj) {
80          Node curNode = it.getNext();
81          LinkedList found = new LinkedList();
82          while (curNode != header) {
83                  if (curNode.element.equals(obj)) {
84                          found.addLast(curNode.element);
85                  }
86                  curNode = it.getNext();
87          }
```

```
88          return found;
89      }
90
91      public pure free LinkedList findAll(readonly Element obj){
92          Iterator it = new Iterator(this);
93          return find(it, obj);
94      }
95
96
97      public String toString() {
98          StringBuffer buffer = new StringBuffer();
99          buffer.append("[ ");
100
101         Node current = header.next;
102         while (current != header) {
103             // the cast to peer is needed since there is no method
104             // StringBuffer#append(readonly Object)
105             buffer.append(current.element.toStringF()).append(" ");
106             current = current.next;
107         }
108
109         buffer.append("]");
110         return buffer.toString();
111     }
112
113     public static void main(String[] args) {
114         System.out.println("concatenate first version:");
115         LinkedList l1 = new LinkedList();
116         l1.addLast(new Element(1));
117         l1.addLast(new Element(2));
118         l1.addLast(new Element(3));
119         System.out.println(l1); // [ 1 2 3 ]
120
121         LinkedList l2 = new LinkedList();
122         l2.addLast(new Element(4));
123         l2.addLast(new Element(5));
124         System.out.println(l2); // [ 4 5 ]
125
126         l1.concatenate(l2);
127         System.out.println(l1); // [ 1 2 3 4 5 ]
128         System.out.println(l2); // [ ]
129
130         System.out.println("concatenate second version:");
131         LinkedList l3 = new LinkedList();
132         l3.addLast(new Element(1));
133         l3.addLast(new Element(2));
134         l3.addLast(new Element(3));
135         System.out.println(l3); // [ 1 2 3 ]
136
137         LinkedList l4 = new LinkedList();
138         l4.addLast(new Element(4));
139         l4.addLast(new Element(5));
140         System.out.println(l4); // [ 4 5 ]
141
```

```
142         Node header = l4.getHeader();
143         l3.concatenate(header.next, header.prev);
144         System.out.println(l3); // [ 1 2 3 4 5 ]
145         System.out.println(l4); // [ ]
146 
147         System.out.println("find:");
148         LinkedList l5 = new LinkedList();
149         l5.addLast(new Element(1));
150         l5.addLast(new Element(2));
151         l5.addLast(new Element(3));
152         l5.addLast(new Element(4));
153         l5.addLast(new Element(3));
154         l5.addLast(new Element(5));
155         l5.addLast(new Element(3));
156         System.out.println(l5); // [ 1 2 3 4 3 5 3]
157         LinkedList l6 = l5.findAll(new Element(3));
158         System.out.println(l6); // [3 3 3]
159 
160     }
161 
162     private static class Node {
163         readonly Element element;
164         peer Node next;
165         peer Node prev;
166         pure Node(readonly Element element, peer Node next, peer Node prev) {
167             this.element = element;
168             this.next = next;
169             this.prev = prev;
170         }
171     }
172 
173     private static class Element {
174         int value;
175 
176         pure Element(int i) {
177                 value = i;
178         }
179 
180         pure boolean equals (readonly Element other) {
181                 if (value == other.value)
182                         return true;
183                 else
184                         return false;
185         }
186 
187         public pure free String toStringF() {
188                 String s = "";
189                 s = s + value;
190                 return s;
191         }
192      }
193 
194 
195     private static class Iterator {
```

```
196         peer LinkedList list;
197         readonly Node current;
198
199         readonly Node getNext() {
200             current = current.next;
201             return current;
202         }
203
204         pure Iterator(peer LinkedList l) {
205             list = l;
206             current = l.header;
207         }
208     }
209
210 }
```

A.2 Merging of Hashtables

```
import java.util.ArrayList;

/**
 * Testing ownership transfer of arrays
 * <p>
 * Expected result: no errors
 *
 * @author schaada
 */

class HashTable {
        uniq peer LinkedList[] hashTable;
        static int size = 5;

        public HashTable() {
                hashTable = new LinkedList[size];
                for (int i = 0; i < hashTable.length; i++) {
                        hashTable[i] = new LinkedList();
                }
        }

        public free peer LinkedList[] getHashTable() {
                peer LinkedList[] res = hashTable;
                hashTable = new LinkedList[size];
                for (int i = 0; i < hashTable.length; i++) {
                        hashTable[i] = new LinkedList();
                }
                return res;
        }

        void merge(free peer LinkedList[] otherTable) {
                for (int i = 0; i < hashTable.length; i++) {
                        hashTable[i].concatenate(otherTable[i]);
                }
        }

        void merge(peer HashTable otherTable) {
                merge(otherTable.getHashTable());
        }

        private static pure int getHashValue(readonly Element e) {
                return ((e.value % size));
        }

        void insert(readonly Element e) {
                int index = getHashValue(e);
                hashTable[index].addLast(e);
        }

        public free String toStringF() {
                StringBuffer buffer = new StringBuffer();
```

```
52          for (int i = 0; i < size; i++) {
53                  String s = hashTable[i].toStringF();
54                  buffer.append(s);
55                  buffer.append("\n");
56          }
57          return buffer.toString();
58          }
59
60          public static void main( String [ ] args ) {
61                  HashTable t1 = new HashTable();
62                  t1.insert(new Element(2));
63                  t1.insert(new Element(3));
64                  t1.insert(new Element(12));
65                  t1.insert(new Element(22));
66                  HashTable t2 = new HashTable();
67                  t2.insert(new Element(32));
68                  t2.insert(new Element(1));
69                  t2.insert(new Element(13));
70                  t2.insert(new Element(23));
71
72                  System.out.println(t1.toStringF());
73                  System.out.println(t2.toStringF());
74                  t1.merge(t2);
75                  System.out.println(t1.toStringF());
76                  System.out.println(t2.toStringF());
77          }
78
79      private static class Element {
80          int value;
81
82          pure Element(int i) {
83                  value = i;
84          }
85
86          public pure free String toStringF() {
87                  String s = "";
88                  s = s + value;
89                  return s;
90          }
91      }
92
93      private static class LinkedList {
94          protected uniq Node header;
95
96          pure public LinkedList() {
97                  Node tmp = new Node(null, null, null);
98                  tmp.next = tmp;
99                  tmp.prev = tmp;
100                 header = tmp;
101         }
102
103         public void addLast(readonly Element o) {
104             addBefore(o, header);
105         }
```

```
106
107         protected void addBefore(readonly Element o, rep[header] Node node) {
108             Node newNode = new Node(o, node, node.prev);
109             newNode.prev.next = newNode;
110             newNode.next.prev = newNode;
111         }
112
113         public boolean isEmpty() {
114             return header.next == header;
115         }
116
117         protected free Node getHeader() {
118             Node result = header;
119
120             header = new Node(null, null, null);
121             header.next = header.prev = header;
122
123             return result;
124         }
125
126         public void concatenate(peer LinkedList other) {
127             if (other.isEmpty()) {
128                 return;
129             }
130
131             Node otherHeader = other.getHeader();
132
133             header.prev.next = otherHeader.next;
134             otherHeader.prev.next = header;
135
136             otherHeader.next.prev = header.prev;
137             header.prev = otherHeader.prev;
138         }
139
140         public free String toStringF() {
141                 String res = new String("[ ");
142
143                 Node current = header.next;
144             while (current != header) {
145                 res += current.element.toStringF() + " ";
146                 current = current.next;
147             }
148
149             res += "]";
150                 return res;
151         }
152     }
153
154     private static class Node {
155         readonly Element element;
156         peer Node next;
157         peer Node prev;
158         pure Node(readonly Element element, peer Node next, peer Node prev) {
159             this.element = element;
```

```
160             this.next = next;
161             this.prev = prev;
162         }
163 
164     }
165 }
```

A.3 AVL Tree

```
1  /**
2   * Testing uniqueness, ownership transfer and local variable type inference
3   * <p>
4   * Expected result: no errors
5   *
6   * @author schaada
7   */
8
9  public class AVLTree {
10     int elem;
11     uniq AVLTree left;
12     uniq AVLTree right;
13     int height;
14
15     private static int height(readonly AVLTree t) {
16         return t == null ? 1 : t.height;
17     }
18
19     AVLTree() {
20     }
21
22     AVLTree(int e, free AVLTree l, free AVLTree r) {
23         elem = e;
24         left = l;
25         right = r;
26     }
27
28         private free AVLTree getLeft(){
29                 AVLTree temp = left;
30                 left = null;
31                 return temp;
32         }
33
34         private free AVLTree getRight(){
35                 AVLTree temp = right;
36                 right = null;
37                 return temp;
38         }
39
40         private void setLeft(free AVLTree left){
41                 this.left = left;
42         }
43
44         private void setRight(free AVLTree right){
45                 this.right = right;
46         }
47
48     public void insert(int i) {
49         left = insert(i, getLeft());    // root is a dummy node
50     }
51
```

```
52     free AVLTree insert(int x, free AVLTree t) { // "this" is parent of real node
53         if (t == null) {
54             t = new AVLTree(x, null, null);
55         } else if(x < t.elem) {
56             t.setLeft(insert(x, t.getLeft()));
57             if (height(t.left)   height(t.right) == 2) {
58                 if (x < t.left.elem)
59                     t = rotateWithLeftChild( t );
60                 else
61                     t = doubleWithLeftChild( t );
62             }
63         } else if( x > t.elem ) {
64             t.setRight(insert( x, t.getRight()));
65             if( height( t.right )   height( t.left ) == 2 ) {
66                 if( x > t.right.elem )
67                     t = rotateWithRightChild( t );
68                 else
69                     t = doubleWithRightChild( t );
70             }
71         } else
72             ; // Duplicate; do nothing
73         t.height = max( height( t.left ), height( t.right ) ) + 1;
74         return t;
75     }
76
77
78     private static int max(int lhs, int rhs) {
79         return lhs > rhs ? lhs : rhs;
80     }
81
82
83     private free AVLTree rotateWithLeftChild(free AVLTree k2) {
84         AVLTree k1 = k2.getLeft();
85         k2.setLeft(k1.getRight());
86         k2.height = max( height( k2.left ), height( k2.right ) ) + 1;
87         k1.height = max( height( k1.left ), k2.height ) + 1;
88         k1.setRight(k2);
89         return k1;
90     }
91
92
93     private free AVLTree rotateWithRightChild(free AVLTree k1 ) {
94         AVLTree k2 = k1.getRight();
95         k1.setRight(k2.getLeft());
96         k1.height = max( height( k1.left ), height( k1.right ) ) + 1;
97         k2.height = max( height( k2.right ), k1.height ) + 1;
98         k2.setLeft(k1);
99         return k2;
100    }
101
102    private free AVLTree doubleWithLeftChild(free AVLTree k3 ) {
103        k3.setLeft(rotateWithRightChild( k3.getLeft() ));
104        return rotateWithLeftChild( k3 );
105    }
```

```
106
107     private free AVLTree doubleWithRightChild(free AVLTree k1 ) {
108         k1.setRight(rotateWithLeftChild( k1.getRight() ));
109         return rotateWithRightChild( k1 );
110     }
111
112
113
114     pure int computeHeight() {
115         int l = (left   == null ? 1 :  left .computeHeight());
116         int r = (right  == null ? 1 :  right .computeHeight());
117         return max(l,r) + 1;
118     }
119
120     pure void test() {
121         System.out.println( "Checking...␣(no␣more␣output␣means␣success)" );
122         if (height != computeHeight())
123                 System.out.println("Error:␣height␣is␣false !");
124
125         int l = height(  left   );
126         int r = height(  right  );
127
128         int diff  = l    r;
129          diff  = ( diff  < 0 ?   diff  :  diff );
130         if ( diff  >= 2)
131                 System.out.println("Error:␣Tree␣is␣not␣balanced!");
132     }
133
134     // Test program
135     public static void main( String [ ] args ) {
136         AVLTree t = new AVLTree( );
137         final int NUMS = 4000;
138         final int GAP = 37;
139
140         for( int i = GAP; i != 0; i = ( i  + GAP ) % NUMS )
141             t. insert (  i  );
142
143         t. left . test ();
144     }
145 }
```