Universe Type System for Scala

Daniel Schregenberger

Master Thesis

Software Component Technology Group
Department of Computer Science
ETH Zurich

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

September 2006 - June 2007

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

Software Component Technology Group
inf | Informatik
Computer Science

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract

Scala combines the object-oriented and functional paradigms to an expressive programming language. It supports the creation of class hierarchies known from object-oriented languages like Java or C# and at the same time allows to model algebraic types through pattern matching, as found in many functional programming languages.

The Universe type system applies the concept of ownership to Java-like languages and allows to control object modification and thereby maintain invariants. It does so by controlling aliasing without restricting it.

This thesis presents an ownership type system for Scala and other Java-like programming languages. It combines support for type genericity with path-dependent types to extend the static expressiveness of the Universe type system.

Acknowledgments

I would like to thank my supervisor Werner Dietl for his contributions and the advices on the report.

And special thanks go to my family and in particular to my brother Stefan.

Contents

Chapter 1

Introduction

This chapter gives a short introduction to both, the Universe Type System and Scala. Its intention is to make our work understandable even for readers not familiar with both of them. Besides that it also gives an outlook to further readings for those interested.

1.1 Universe Types

The Universe type system [10], an extension of the Java programming language, represents an ownership type system enforcing the owner-as-modifier discipline. This discipline does not restrict aliasing, but allows owners to control state changes of their owned objects. This property can for example be used to maintain invariants and to support reasoning about various program properties.

1.1.1 Ownership

Ownership type systems partition the object store and thereby establish a hierarchical relation between the objects. The fundamental concept of ownership assigns each object at most one *owner*. The set of all objects with the same owner is called *context*. The set of objects with no owner is called the *root context* and the tree structure evolving from these rules is called the *ownership topology*.

1.1.2 Ownership modifiers

In the Universe type system static type rules establish the ownership topology and enforce the owner-as-modifier discipline. In order to do so, types are annotated with ownership modifiers. Such ownership modifiers in the Universe type system are always relative to some object, which we are calling the viewpoint object or simply the *viewpoint*.

Using `this`, the current instance of the enclosing class, as the viewpoint, references to objects sharing the same owner with `this` are expressed by the `peer` modifier and references to objects owned by `this` by the `rep` modifier. Finally the `any` modifier can be used to refer to any arbitrary object without knowledge of its owner. Figure 1.1 illustrates the modifiers and the relations between objects.

Since ownership modifiers are viewpoint dependent but contexts are absolute, different ownership modifiers may, depending on the viewpoint, map to the same context.

1.1.3 Subtyping for ownership modifiers

The definition of ownership modifiers suggests establishing some kind of subtype relation between them. Figure 1.2 illustrates this relation. Keep in mind that ownership modifiers are properties of the reference, not the object itself, and they are thus viewpoint dependent.

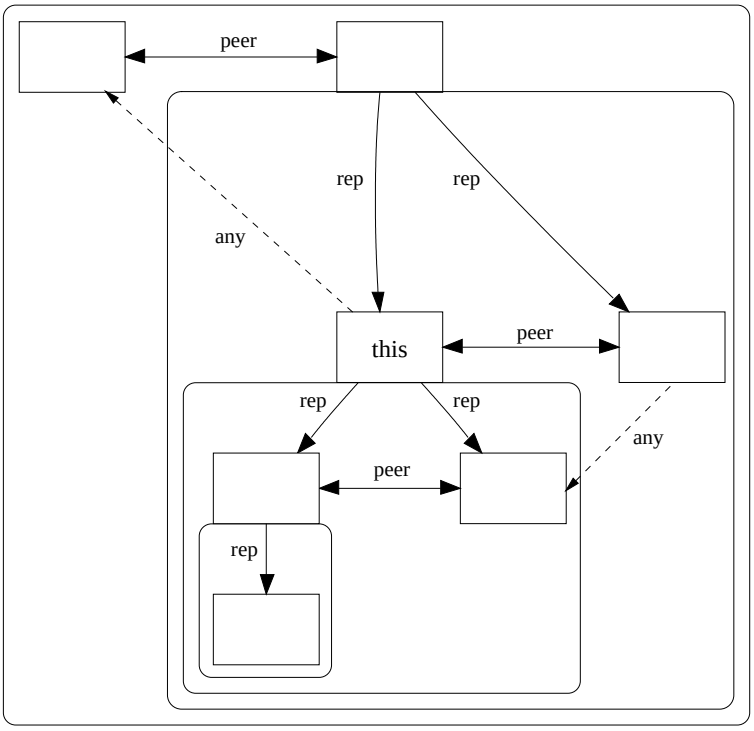


Figure 1.1: Visualization of the ownership relations in an object structure. Objects are represented by boxes and sit atop the context (rounded boxes) of objects they own. Ownership modifiers are attributes of the reference. For example an object pointed to by a `rep` reference (an arrow in the visualization) is `rep` to the object holding the reference (the source of the arrow). Note that only `any` references are allowed to cross arbitrary context boundaries.

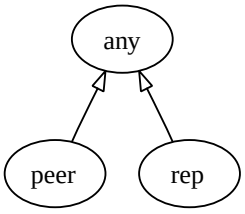


Figure 1.2: Type hierarchy for ownership modifiers.

1.1.4 Aliasing control

Unlike other ownership type systems, Universe Types do not apply the owner-as-dominator discipline to control referencing. Systems enforcing the owner-as-dominator discipline restrict aliasing by requiring every reference to an object `o` having to go through `o`'s owner. Instead the Universe type system enforces the less restrictive *owner-as-modifier* discipline, which allows any object to reference some object `o`, but ensures that only reference chains passing through `o`'s owner may be used to modify `o`. This allows owners to control state changes of their owned objects and thus maintain invariants.

According to [8] the owner-as-modifier discipline is enforced by disallowing modifications of objects through `any` references. Expressions of an `any` type may only be used as receiver of field reads and to call side-effect free methods. To check this property, side-effect free methods are required to be annotated with the keyword `pure`.

Note that like Aldrich and Chambers [1] we consider ownership type systems to be twofold: firstly the concept of ownership establishes the aforementioned ownership topology and secondly aliasing control is applied to enforce encapsulation. We call systems without aliasing control *topological type systems* and consider the aliasing restrictions to manifest themselves in an overlay system named the *encapsulation type system*.

1.1.5 Generic Universe Types

Generic Universe Types (GUT) [8] presents the first type system that combines the owner-as-modifier discipline with type genericity. A type in [8] is either a type variable or consists of an ownership modifier, a class name, and possibly type arguments. The ownership modifier expresses object ownership relative to the current receiver object `this`. The modifiers of type arguments also express ownership relative to the current `this` object and not relative to the instance of the generic class that contains the argument modifier.

Note that when we are referring to the Universe type system, we are actually referring to Generic Universe Types as its latest incarnation.

1.2 Scala

Scala is a modern programming language which combines object-oriented and functional programming in an elegant way. It is a pure object-oriented language in the sense that every value is an object and its mixin-based composition mechanism provides a clean alternative to the multiple-inheritance models of other languages. By supporting higher-order functions, currying, function nesting, pattern matching, and anonymous functions it is also a functional language.

Scala code is compiled to either Java bytecode or .net assemblies. The integration with these platforms is very tight. For example Java classes can be used from Scala code and vice versa. This allows Scala programs to reuse the existing Java libraries and keeps the price for switching from Java to Scala low.

In the following we give a short motivation for combining Universe Types with Scala and a short introduction to the Scala syntax. While this should be sufficient to understand our type system, we encourage to read some of the various documents, papers, and web pages about Scala. A good starting point is the official Scala web page[1]. To get a high-level overview over Scala we recommend the paper by Odersky et al. [19] and if more in-depth information about a specific feature is required, the Scala language reference [18] is a good source.

There are also several papers [17, 24, 20] that serve as an introduction for programmers willing to give Scala a try. Since people are usually more familiar with object-orientation and Java than with the functional concepts Scala supports, most of them are more or less focusing on advertising the functional features. We are sharing the opinion of the authors that the transition from Java to Scala for the object-oriented features is straightforward. However, the real elegance and power

[1] `http://www.scala-lang.org/`

of Scala lies in its functional features, which usually need some time to get accustomed to. Be sure to check the community page of the Scala website. There are more and more people on the Internet sharing their experiences with Scala, giving hints and tips.

1.2.1 Why Scala?

The main objective of our work was to create a system that combines path-dependent types with Universe Types. Since Scala supports path-dependent types and, among other things, also the very interesting concept of definition-site variance annotations for type parameters, we decided to use the Scala language to base our system on.

1.2.2 Short introduction to the Scala syntax

The syntax of Scala is not fundamentally different from that of Java and for someone familiar with Java it should be possible to understand basic Scala programs. Some Scala code, however, may require a bit of knowledge in functional programming to fully understand it, but since Universe Types are all about objects and references, the lack of a functional background should be no problem for understanding this report.

```scala
class Syntax[T] (init: T) {
    val constant: T = init
    var variable = init


    def printMessage (m: String) = {
        Console.println(m)
    }
}
```

Scala

```java
class Syntax<T> {
    final T constant;
    T variable;

    Syntax (T init) {
        constant = init;
        variable = init;
    }

    void printMessage (String m) {
        System.out.println(m);
    }
}
```

Java

Figure 1.3: Comparison of Scala and Java syntax.

Figure 1.3 shows a side-by-side comparison of the Scala and the Java syntax by means of a small example program. Most of the differences are only notational: Scala uses square brackets to delimit type parameters instead of the angle brackets used by Java. In Scala each definition is prepended by a keyword such as `val` for values (`final` variables in Java), `var` for variables and `def` for methods. Types are specified after the identifier, separated by a colon. For method return values and local values or variables types may even be omitted, in which case the compiler automatically infers them. Probably the biggest difference in this example is the fact that in Scala the class body represents the default constructor.

1.3 Notation and naming conventions

We are adopting the Scala way of naming things as opposed to the Java way used in previous Universe related papers. For example we use the term *parameterized types* for what the Java community calls generic types.

Like Generic Universe Types we are using tuples of an ownership modifier and a type, as in `any List[any Object]`, for the notation.

1.4 Outline

Chapter 2 motivates the use of path-dependent types to extend the Universe type system and provides and informal overview of the main concepts of our type system. Chapter 3 then presents a first formalization of the type system and Chapter 4 discusses some of its aspects. In Chapter 5 related work is discussed and Chapter 6 concludes and gives an outlook for possible directions of future work.

Chapter 2

Informal Overview

In this chapter we motivate the introduction of path-dependent types to the Universe type system and provide an informal overview of the main concepts of our type system.

2.1 Path-dependent types

The idea of path-dependent types is closely related to the concepts of virtual types and virtual classes. While the idea of virtual classes has been around for a while, the theoretical foundations have taken a bit longer to evolve. In fact only recently a virtual class calculus [14] has proven them to be not as inherently unsafe as they were assumed to be.

Virtual classes are in many ways similar to virtual methods. They are defined as attributes of classes, may be redefined within subclasses, and are accessed with dynamic binding through object instances and not statically, like for example inner classes in Java. This means the actual value of a virtual class is generally not known at compile time but depends on the identity of the object it is accessed through. A virtual class therefore only specifies a well-defined type if it is accessed through an immutable reference, or more generally, an immutable expression. Such immutable expressions are better known as paths and the types introduced by virtual classes as path-dependent types.

2.2 Universe Types are viewpoint-dependent

When introducing Universe Types, we have emphasized that universes are viewpoint dependent. So far we have implicitly assumed the viewpoint to be the current instance `this` of the enclosing class. More explicitly one could therefore write `this.peer` instead of the shorter `peer` and `this.rep` instead of `rep`. The idea of using paths to introduce other viewpoints than `this` is now just a step away.

Similar to the additional typing power introduced by parameterized types, path-dependent Universe Types have the potential to increase static type safety of programs using Universe Types. And since there are already a number of systems [5, 21, 6, 14] using virtual classes or virtual types to establish family polymorphism [12], path-dependent types suggest themselves to increase the static type safety of ownership type systems.

The concept of family polymorphism is very similar to what can be expressed with an ownership type system. Family polymorphism on object level groups objects to families. For example a list and all its nodes form a family. Nodes from different lists belong to different families and are distinguished by the identity of the list object, the so-called *family object*, to prevent (accidental) mixing of nodes of different lists. Yet if the family identity is not of interest, for example in a pure method, nodes of different lists can be processed by simply typing them with their common supertype.

2.3 Path-dependent types in Scala

Since Scala already includes support for path-dependent types, we evaluated the possibility of using them to establish the ownership topology. Path-dependent types in Scala are introduced by type members accessed through paths. According to [18] a path in Scala may be the empty path ϵ, C.`this`, C.`super` (where C references a class) or $p.x$, where p is a path and x a stable member of p. A member is stable if it is introduced by a value, singleton object or package definition. For an example consider the definition on line 1 in Figure 2.1, which introduces `T` as a type member of class `Cell`.

Our initial idea was that a class would simply define type members for all types it would be using in its representation and then only use the dependent type introduced by this type member for declarations. Figure 2.1 shows a sketch of this idea using inner classes. Our assumption was that a type $p.T$ would be a subtype of the type referenced by it and only valid if accessed via a path p. Unfortunately it turned out that path-dependent types in Scala are, due to their relation to virtual classes, too heavily linked to inner classes for this idea to work out.

The main use for type members in Scala is the definition of abstract datatypes and not path-dependent types. For example the Featherweight Scala calculus [6], which captures amongst other Scala core constructs also dependent types, has a rich syntax of types combining both, path-dependent types and abstract datatypes. It defines a type to be virtual only if the type member introducing it is abstract. If the type member is bound to a specific type, it represents this type and is indeed nothing more than a mere alias.

```
0 class Cell {
    type T = Node
    var value: T = new T

    class Node
5 }

  object inner extends Application {
    val cell = new Cell
    val item = new Cell
10
    cell.value = item.value   // error: type mismatch
    item.value = cell.value   // error: type mismatch

    cell.value = new cell.T
15  item.value = new item.T
  }
```

Figure 2.1: Path-dependent types in Scala using inner classes.

Notice the type mismatch between the types of `cell.value` (`cell.T`) and `item.value` (`item.T`) in Figure 2.1. Expressing the same without using inner classes, as in Figure 2.2, fails since, unlike our expectations, the types of `cell.value` and `item.value` are compatible. They are in fact, as explained above, aliases for type `Node`.

However, our goal was to enable the programmer to freely decide what classes he wants to group to a context and thus allowing a higher degree of reuse without losing type safety, similar to the idea of generics. Since path-dependent types in Scala are to closely related to inner classes and fail to ensure type safety on a level required for an ownership type system, we decided to build our own system of path-dependent types. Our experience suggests that a separate system lets us better focus on ownership than piggybacking ownership types on Scala's path-dependent types would.

```
class Cell {
    type T = Node
    var value: T = new T
}

class Node

object path extends Application {
    val cell = new Cell
    val item = new Cell

    cell.value = item.value   // ok
    item.value = cell.value   // ok

    cell.value = new Node     // ok
    item.value = new Node     // ok
}
```

Figure 2.2: Path dependent types in Scala using arbitrary classes.

2.4 Viewpoint adaptation

Since universes are viewpoint dependent, Generic Universe Types perform viewpoint adaptations to compute the combined type of expressions such as field selections, method parameters and return values. To compute the combined type of a field expression `c.f` for example, the type of `f`, which is relative to its enclosing object instance `c`, has to be adapted to the viewpoint `this`.

The viewpoint adaptation has to be re-evaluated and adapted for path-dependent Universe Types, which we are doing in this section.

2.4.1 Viewpoint adaptation in Generic Universe Types

The basic viewpoint adaptation function used in GUT is depicted in Figure 2.4(a). If for expression `c.f` the receiver `c` equals `this`, then the adapted type equals the field type since it already is relative to the viewpoint `this`. If the field is `peer`, the object referenced by it has the same owner as the receiver used to access it (here `c`). If `c` is `peer` as well it has the same owner as `this` and thus `c.f` has the same owner as well. And if `c` is `rep` it is owned by `this` and so is `c.f`. For any other case it is impossible to statically express the exact relation between `this` and `c.f` in the Universe type system. Therefore the combined type for all remaining cases is `any`.

In accordance with the owner-as-modifier discipline no updates are possible if the universe of the receiver is `any`, since a reference to an object in an arbitrary context may not be used to update fields. Along the same lines only `rep` fields of `this` are modifiable. It can easily be seen that all cases in Figure 2.4(a) which are not marked to be forbidden by the type rules can be allowed without breaking type safety. Thus type safety with regards to the ownership topology is in GUT ensured by the type rules enforcing the owner-as-modifier discipline. The viewpoint adapted type alone does not indicate whether modifications are permitted or not.

2.4.2 Separating encapsulation and topology

In order to keep focused on path-dependent types we decided to build our system step by step by first establishing the ownership topology without any aliasing control. Enforcing encapsulation by means of the owner-as-modifier discipline is then done in a second step towards the final system.

Decoupling topology and encapsulation helps us evaluating the impact of the path-dependent types on the Universe Type System and allows experimenting with novel ideas and concepts regarding the encapsulation policy.

To separate the establishment of the ownership topology from the encapsulation enforced by the owner-as-modifier discipline we need to drop the restrictions in the type rules. This enables us for example to use an `any` reference to assign to an `any` field, which does not corrupt the ownership topology since such a field does not care about the context of the object it references. On the other hand we have to disallow updates of `peer` or `rep` fields on `any` references since such fields care about the context of the objects they reference but the `any` type of the receiver does not allow assignments conforming to such field types.

Unfortunately the removal of the restrictions imposed by the owner-as-modifier discipline not only invalidates the encapsulation properties, but also breaks type safety with regards to the ownership topology. Therefore the viewpoint adaptation function has to be adjusted to indicate topological type safety of modifications all by itself.

```
0 class C {
    val f: peer Object
  }

  val c: any C = new peer C
5 val a: any Object = c.f

  // ...
  c.f = a    // compile time error
```

Figure 2.3: Viewpoint adaptation for field selections.

As an example consider the type of the expression `c.f` in Figure 2.3. According to the viewpoint adaptation function in GUT the combined type is `any` and while read accesses are always allowed for Universe Types, assigning a value to `c.f`, as attempted on line 8, cannot be allowed since it would break type safety. For example one could try to assign it a `rep` reference. But since, as explained above, we have no knowledge about the owner of `c.f`, it is not safe to assign it a reference to an object owned by `this`. This means for viewpoint adapted types being `any` we have to differentiate between those with field type `any`, in which case assignments can be allowed, and those with a different field type.

	peer	rep	any
this	peer	rep	any
peer	peer	any*	any
rep	rep	any*	any
any	any*	any*	any*

(a) In GUT.

	peer	rep	unknown	any
this	peer	rep	unknown	any
peer	peer	unknown	unknown	any
rep	rep	unknown	unknown	any
unknown	unknown	unknown	unknown	any
any	unknown	unknown	unknown	any

(b) Adapted for Path-dependent Universe Types.

*Assignments forbidden through type rules.

Figure 2.4: The viewpoint adaptation function.

In order to achieve both, being able to assign `any` fields through arbitrary references and nevertheless protect type safety for `peer` and `rep` fields, we are introducing the `unknown` modifier as a new ownership modifier. The viewpoint adaptation function is adjusted to return `any` whenever the field type is `any` and `unknown` for all cases where it is impossible to statically determine the relation between the current viewpoint `this` and the object referenced by a field selection expression `c.f`. To protect type safety, field updates are then only permissible if the viewpoint adaptation for the left-hand side does not yield `unknown`. Along the same lines methods with an expected parameter universe being `unknown` can not be called. With this restriction we can safely

consider `unknown` types to be subtypes of their respective `any` type.

Figure 2.4(b) depicts the adapted viewpoint adaptation function for our topological type system and Figure 2.5 shows the updated type hierarchy including `unknown`. The updated viewpoint adaptation function clearly shows that the purpose of the `unknown` modifier is to ensure that type safety with regards to the ownership topology is not violated. And the type hierarchy indicates that a reference annotated with `unknown` may, similar to one annotated with `any`, refer to objects in arbitrary contexts.

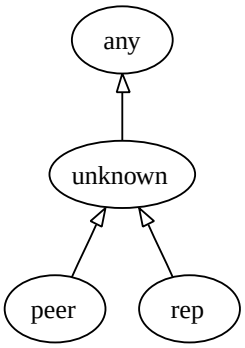


Figure 2.5: Updated type hierarchy for ownership modifiers.

Since the main use of the `unknown` modifier is protection of type safety, we enforce useful uses of it. So since the `any` modifier can always be used to annotate references to arbitrary objects, we do not allow `unknown` to be used as main modifier.

2.5 Limited covariance for parameterized types

Abandoning the owner-as-modifier discipline also makes it necessary to revisit the limited covariance and viewpoint adaptation problem described in Generic Universe Types [8].

While subtyping with covariant type arguments is in general not statically type safe, the owner-as-modifier discipline allows for a limited form of covariance without additional runtime checks. As an example consider the type `peer List[rep Object]` of value `l` in Figure 2.6. According to the limited covariance in GUT `any List[any Object]`, the type of value `a`, would be an admissible supertype, since not only the element modifier is `any` but also the main modifier. Thus for reference `a` any modification of the list that could possibly break type safety is prohibited and assigning `l` to `a` as done on line 1 can be permitted.

```
0 val l: peer List[rep Object] = new peer List[rep Object]
  val a: any List[any Object] = l
```

Figure 2.6: Limited covariance for an encapsulating type system as proposed by GUT.

But for a purely topological type system without encapsulation this does not hold and modifying the main modifier to `any` whenever the modifier of a type argument changes to `any` does not yield an admissible supertype. So for our system `any List[any Object]` is not a supertype of `peer List[rep Object]`. Simply because a reference of this type would still allow modifications of the list and therefore type safety could be broken by putting an object of an arbitrary context into the list. But the referenced list might, and for this case does in fact, care about the actual context of the objects it holds. Even if we do not know which context it is, it is a specific one and we are obliged to only add objects of this specific context to the list.

However, we do not have to resort to invariant handling of type parameters. The `unknown` modifier, when used to annotate type arguments, allows us to express that we have no knowledge

about the context of the elements of a parameterized type. This of course implies that we cannot allow modification of the elements or addition of new elements.

While for our system collections of `any` objects may hold references to objects of any number of contexts, collections of `unknown` objects are (in general) considered to hold objects of one specific, yet unknown context, hence the name. See figure 2.7 for an illustration of the limited covariance for parameterized types allowed by the `unknown` modifier.

```
val l: peer List[rep Object] = new peer List[rep Object]
val k: rep List[peer Object] = new rep List[peer Object]
var u: any List[unknown Object] = l
var a: any List[any Object] = l    // compile time error
u = k
a = k    // compile time error
u = a
```

Figure 2.7: Limited covariance for a topological type system using the `unknown` modifier.

While, for the reasons explained in the previous section, we are not allowing the `unknown` modifier to be used as main modifier, it is allowed to be used to annotate type arguments. It is, however, not possible to create a new instance of a parameterized type if one of the main modifiers of its direct type arguments is `unknown`. This restriction applies since it would be impossible to create elements for such a type. If a type with elements in arbitrary contexts is wanted, the `any` modifier should be used as element modifier instead. Figure 2.8 shows some examples illustrating these restrictions.

```
val x = new unknown Object          // compile time error
val y = new peer List[unknown Object]  // compile time error

/* a list containing lists with unspecified element type */
val z = new rep List[any List[unknown Object]]   // ok
z.add(new peer List[rep Object])   // ok
z.add(new rep List[peer Object])   // ok
z.add(new rep List[any Object])    // ok
```

Figure 2.8: Examples for using the `unknown` modifier.

2.6 Main concepts

In this Section we explain the main concepts of our path-dependent Universe Types with informal examples.

2.6.1 Additional ownership modifiers

We are adding three additional ownership modifiers. The `up` modifier allows objects to hold a qualified reference into their owners context. Figure 2.9 revisits the object structure from Figure 1.1 and illustrates the usage of the `up` modifier as well as the virtual `owner` field referencing an objects owner.

The `reps` modifier allows to reference the `rep` context and any context below it in the topology. In other words it is a shorthand for sequences of `rep` modifiers and therefore subsumes all paths to objects in contexts transitively owned by `this`. The third new modifier, `ups`, carries a very similar idea. It is a shorthand for sequences of `up` modifiers and allows to reference all objects in contexts along the context topology path from `this` to the root object. See Figure 2.10 for a

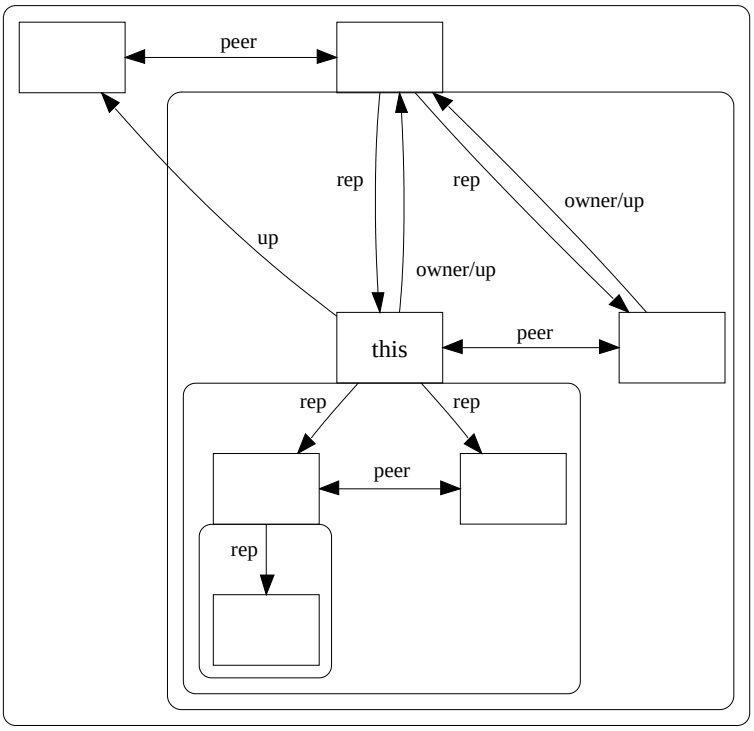


Figure 2.9: Visualization of the `owner` field and the `up` modifier in the object structure from Figure 1.1.

visualization of the `reps` and `ups` modifiers and Figure 2.11 for the updated type hierarchy for ownership modifiers.

2.6.2 Paths

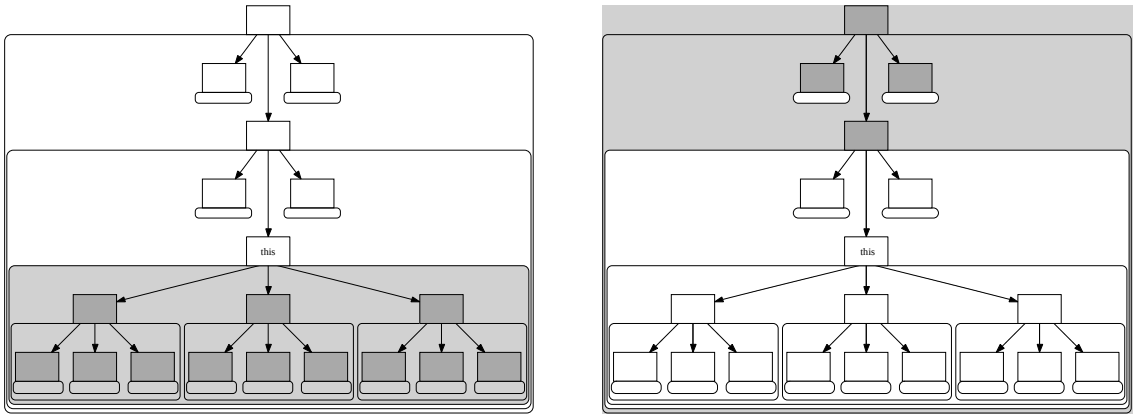
Every *path* starts with a local value or variable, the `this` reference, a singleton object, or a value parameter[1]. It is then followed by any number of field selections and a sequence of context modifiers: $Start.\overline{Field}.\overline{Context}$. For simplicity we are not allowing packages or super references in paths. They are not of special interest to us but could be added easily if needed.

To avoid confusion we are using the explicit form with leading `this` for the rest of this thesis. And we are also adjusting the naming conventions to avoid confusions between the simple ownership modifiers in GUT and their use as path elements in our system. The `peer`, `rep`, `up`, `reps`, and `ups` modifiers are therefore referred to as *context modifiers* from now on. And for our system paths form, together with the `unknown` and `any` modifiers, the class of ownership modifiers which we are referring to as *universes*.

2.6.3 Different forms of path types

We partition paths into three classes, namely stable, exact, and loose paths. A path is considered to be *stable*, if it starts with an immutable value, contains only immutable field selections and no trailing contexts. This corresponds to the notion of stable paths used in Scala. *Exact* paths consist of a stable prefix followed by exactly one of the `peer`, `rep` or `up` modifiers. Any path that is neither stable nor exact is considered to be *loose*. This includes all paths containing a `reps` or `ups` modifier or more than one context modifier.

[1] Value parameters, as opposed to type parameters, are simply the usual method parameters. In Scala method parameters programs are immutable, while in Java a method may modify its parameters and use them just like local variables.



(a) `reps` (b) `ups`

Figure 2.10: The ownership modifiers `reps` and `ups`. A bright shade of grey is used to denote contexts and a darker shade for objects contained by the respective modifier. Contexts and objects remaining white are not covered by the modifier.

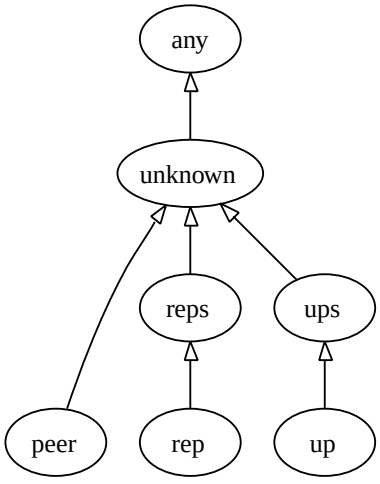


Figure 2.11: Updated type hierarchy for ownership modifiers.

Loose paths, as opposed to exact paths, describe not a single context but a family of contexts. For example the loose path `this.peer.rep` describes all contexts owned by peers of `this`, including the context owned by `this` itself.[2] Variable `h` in line 11 of Figure 2.12 is annotated with this loose path. In this environment, it subsumes the paths `this.rep`, `f.rep` and `g.peer` and therefore is a supertype of them. More generally variable `h` may, as its type indicates, refer to any `rep` object of the current `this` reference or its peers.

```
0 class D

  class E {
    var x: this.peer D = new this.peer D
    var y: this.rep D = new this.rep D
5 }

  class C {
    val f: this.peer E = new this.peer E
    val g: this.rep E = new this.rep E
10
    var h: this.peer.rep Object = g
    h = g.x
    h = f.y
  }
```

Figure 2.12: Loose path example.

Generally stable paths identify an object, exact paths identify one specific context and loose paths describe families of contexts.

2.6.4 Viewpoint adaptation for path-dependent Universe types

Path-dependent types increase the precision of the viewpoint adaptation function used to type expressions. Figure 2.13 revisits the example from Figure 2.3 to illustrate this for a field selection.

```
0 class C {
    val f: peer Object
  }

  val c: any C = new peer C
5 val a: c.peer Object = c.f

  // ...
  c.f = a    // ok
```

Figure 2.13: Type combination for field selections

Instead of resorting to `any` to type value `a`, path-dependent types allow to use the more expressive `c.peer`. Generally the path-dependent viewpoint adaptation for a field selection expression `c.f` is performed by substituting the receiver expression (here `c`) for the leading `this` in the field type (here `this.peer`). If `c` is a mutable expression the resulting degenerate path is only an intermediate result. The next section gives more insight on the handling of paths containing mutable components.

[2] The context owned by `this` is included according to the subtype relation for ownership modifiers $<:_u$. See Section 3.1.8 for details.

2.6.5 Paths containing mutable fields

Because we are only using paths to identify the context of an object and not the object itself, it is safe to allow paths containing mutable fields to some extent. We believe it improves the readability of a program when expressions which are used relative to some object `o` can also be typed relative to `o`. Even if `o` is mutable or there would be an equivalent path without mutable components. Mutable path elements are therefore allowed for convenience.

However, internally all user-specified paths are normalized and the resulting paths contain no more mutable parts but consist of a stable prefix and one or more trailing context modifiers. The actual type system therefore does not have to deal with mutable paths, they are simply syntactic sugar for the programmers.

```
0  class E

   class C {
     var f: this.peer E
     var g: this.rep E
5
     def m (a: this.peer E) : this.rep E = {
       // ...
     }

10   def foo (x: this.rep E) = {
       // ...
     }
   }

15 object mutable {
     var c: this.rep C = new this.rep C
     val ret: c.rep E = c.m(c.f )          // normalized universe: this.rep.rep

     c.g = ret      // compile time error
20   c.foo(ret)     // compile time error
   }
```

Figure 2.14: Motivation for mutable paths.

Figure 2.14 shows an example that further motivates the need for mutable paths. If we would disallow mutable paths altogether, the method call on line 17 would be impossible, since the expected type for argument `a` depends on the receiver of the method call, which in this case is the mutable reference `c`. This call is, however, type safe and perfectly legal. The normalization of the argument universe `c.peer` yields `this.rep`, which contains no mutable parts anymore. In fact the normalization of the universe of `c.f` to `this.rep` does not loose any information. It is equivalent to a simple universe combination as discussed in Section 2.4.

The universe of the method return value `c.rep`, on the other hand, is normalized to the loose path `this.rep.rep` and so is the universe of value `ret`. Even though the normalized universe of `c.g` is `this.rep.rep` as well, the assignment on line 19 is not type safe since the path `this.rep.rep` does not identify the exact context of `c.g`. Along the same lines the call to method `foo` on line 20 cannot be allowed either.

To avoid confusions originating from normalized paths, implementations of our type system should issue a warning to the programmer if the normalization result differs from the original path. In order to keep the amount of such warnings on an acceptable level, only normalizations loosing information should be reported. If the normalization was lossless the warning is not mandatory. Figure 2.15 shows an example for each.

```
var c: this.peer C = new this.peer C

val x: c.peer C  // lossless normalization to this.peer
val y: c.rep C   // lossy normalization to this.peer.rep — issues a warning
```

Figure 2.15: Example of a lossless and a lossy normalization of mutable paths.

2.6.6 Path normalization and conventions

As explained above paths specified by programmers are normalized in order to allow mutable paths. Normalization also ensures paths are valid with respect to the following restriction: paths given by programmers need to end with a context modifier. In other words they must not be stable. This helps indicating that universes are used to identify the context of an object. A universe describes a group or family of objects, not a single object as stable paths do.

Exact paths represent the most common form of ownership annotations for our path-dependent universe type system. In the form of `this.peer` and `this.rep` they also subsume the `peer` and `rep` modifiers known from previous Universe type systems. For convenience we are even allowing paths to omit the leading `this`. Paths starting with a field name or context modifier are implicitly assumed to have `this` as start value and the empty path is implicitly expanded to the default path, usually `this.peer`.

2.6.7 Path aliasing

Path types introduce a new source for aliasing problems. Since objects of an equivalent or compatible type do not have to be of syntactically equivalent types, type aliasing may occur. Figure 2.16 shows an example for the type aliasing introduced by paths. The assignment on line 9 is correct

```
class D

class C {
  var d: this.peer D = new this.peer D
}

// ...
val c: this.rep C = new this.rep C
val d: c.peer D = c.d
var e: this.rep D = d
c.d = e
```

Figure 2.16: Path aliasing example.

despite the syntactically different types. The same holds for the field update on line 10. While for the former assignment one could argue that `this.rep` has to be a supertype of `c.peer`, the later clearly cannot be solved through subtyping and indicates that the two types are in fact equivalent. The type system we are presenting is able to detect this kind of type equivalence by simplifying each path to its most simple form. In our example the most simple universe for value `d` would be `this.rep`, since replacing `c` in `c.peer` by its universe `this.rep` yields `this.rep.peer` which can be simplified to `this.rep`. More details on path simplification are given in Section 3.1.7.

Path simplification forms an important part of our system. It ensures path aliasing is low which prevents the programming from getting too tedious and therefore keeps the system functional. Without path simplification, the left-hand side for the assignment on line 10 would be typed with `c.peer`, which has to be considered incompatible with `this.rep`, the universe of variable `e`, and the assignment would be rejected. It is, however, important to notice that path simplification

retains the exactness of the original path and does not loose any information. Thus no warning to the user has to be issued.

2.6.8 Subtyping

Loose paths introduce a whole lot of new types and relations between them. Previously we have already seen that `this.rep` is a subtype of `this.peer.rep`, `reps` is a shorthand for sequences of `rep` modifiers and likewise `ups` for sequences of `up` modifiers. Additionally a path containing other stable components besides `this` is a subtype of the path computed by loosening it.

```
0 class E
  class D {
    var f: this.rep E = new this.rep E
  }
  class C {
5   val d: this.rep D = new this.rep D

    var x: this.d.rep E = this.d.f
    var z: this.rep.rep E = x          // ok
  }
```

Figure 2.17: Loose supertypes.

For example path `this.d.rep`, the path of field `x` in Figure 2.17, is a subtype of `this.rep.rep` as the definition of field `z` indicates. This can be established since `this.d.rep` identifies the context owned by the object field `d` references and since this object is again owned by `this`, the current instance of class `C`. Field `z` therefore may reference any object owned by an object owned by `this`. Computing loose superpaths is discussed in detail in Section 3.1.6.

Combining the rule that every path is a subtype of its loosened form with the subtyping rules for `reps` modifiers leads to a rich subtyping hierarchy. Figure 2.18 depicts two code fragments and the respective type hierarchies showing the possible universes to type field `z`. While for Figure 2.18(a) there are quite a lot of supertypes for the most exact type `this.d.f.rep`, for Figure 2.18(b) the most exact type `this.d.f.peer` can be simplified to `this.peer`. It is possible to use universes like `this.d.f.peer` or `this.d.peer` to annotate `z`. They are, however, simplified to `this.peer` by the type system to reduce path aliasing as explained in the previous section and are therefore not listed in the type hierarchy.

Note that the type hierarchies shown in Figure 2.18 are not complete since the number of supertypes, from a static perspective, is not finite. Consider for example that `this.peer` is a subtype of `this.up.rep`, which is again a subtype of `this.up.up.rep.rep` and so on. This holds due to the fact that `this.up` captures the owner of `this` and its peers and `this.up.rep` therefore captures all objects owned by the owner of `this` and its peers, which includes `this` and its peers.

2.6.9 Methods with dependent parameters

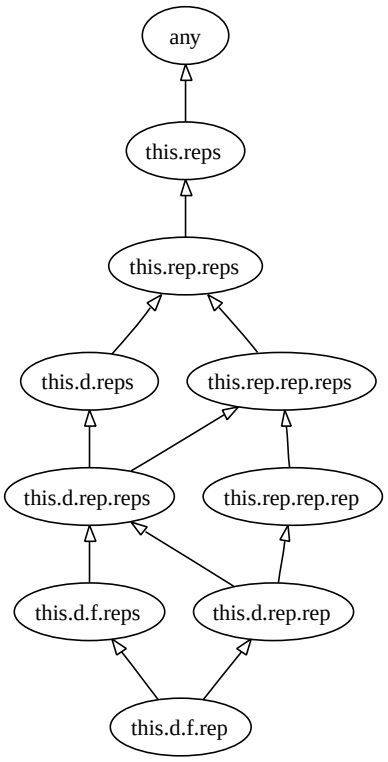
When discussing paths containing mutable fields we have already seen that path-dependent Universe types allow to call methods taking `rep` parameters on other receivers than `this`. Additionally it is possible to type method parameters relative to preceding parameters. Along the same lines the type of the return value may depend on any of the parameters.

Figure 2.19 shows an example for such a method. Note that in this case it is safe to allow the method call even for mutable receivers, as the calls on lines 12 and 13 illustrate. The argument corresponding to parameter `a`, however, has to be immutable. Otherwise it is impossible to provide an argument for parameter `b`. Therefore the call on line 14 fails despite the immutable receiver `c`.

```
class E
class D {
  val f: this.rep E
  val g: this.f.rep E
}

class C {
  val d: this.rep D = new this.rep D

  val z: ??? E = this.d.g
}
```

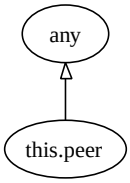


(a)

```
class E
class D {
  val f: this.peer E
  val g: this.f.peer E
}

class C {
  val d: this.peer D = new this.peer D

  val z: ??? E = this.d.g
}
```



(b)

Figure 2.18: Example type hierarchies

```
class C {
  val x: this.rep Object = new this.rep Object

  def foo (a: this.peer C, b: a.rep Object) = {
    // ...
  }
}

val c: this.rep C = new this.rep C
val d: this.rep C = new this.rep C
var z: this.rep C = new this.rep C

c.foo(d, d.x)
z.foo(d, d.x)
c.foo(z, z.x)    // compile time error
```

Figure 2.19: Example of a method where one parameter depends on another parameter.

2.6.10 Parameterized types

In Section 2.5 we have already seen that the `unknown` modifier allows a limited form of covariance for parameterized types. Additionally we make use of Scala's support for variance annotations for type parameters to allow full co- and contravariance. This is possible because Scala ensures that covariant type arguments are never used in contra- or invariant position. See Section 4.5 of the Scala language reference [18] for more details.

Figure 2.20 shows a small example of a covariant parameterized type `D`.

```
class D[+X] {
  // ...
}

class C {
  var f: this.peer D[this.rep Object] = new this.peer D[this.rep Object]
}

// ...
var c: this.rep C = new this.rep C

val x: this.rep D[this.rep.rep Object] = c.f    // ok
```

Figure 2.20: Example of a parameterized type with covariant parameter `X`.

The limited covariance from Section 2.5 is nevertheless still useful to improve the flexibility for invariant type arguments.

2.6.11 Visibility

To ensure a type can be interpreted everywhere an expression of this type appears, we impose the restriction that paths may only depend on fields of the same or a higher level of accessibility as their own. This means private fields, as well as parameters and return values of private methods, may depend on private, protected or public fields of its containing class instance, whereas public ones may only depend on other public fields.

Chapter 3

Path-dependent Universe Types

This chapter presents a first formalisation of our type system. The next section defines the syntax of our type system and explains the type rules and predicates used by them. The presented rules do not form a complete formalization, since this would have been out of scope for this thesis. Complete formal specification and a type safety proof is left to future work. The current kind of semi-formalization, however, turned out to be a much easier and better understandable way to describe the type system than with just informal text.

3.1 Static type system

This Section covers the static type system for path-dependent Universe Types. Note that the presented rules do only cover non-parameterized types. We are extending this type system to parameterized types in Section 3.2.

3.1.1 Syntax

T	$\in$	$Type$	::=	$Univ\ SType$
C	$\in$	$SType$	::=	Scala type conforming to `scala.AnyRef`
u	$\in$	$Univ$	::=	$UPath$ | `unknown` | `any`
p	$\in$	$UPath$	::=	$OPath.\overline{Context}$
id	$\in$	$OPath$	::=	$Start.\overline{Field}$
s	$\in$	$Start$	::=	Val | Var
c	$\in$	$Context$	::=	`peer` | `rep` | `up` | `reps` | `ups`
f	$\in$	$Field$	::=	declared fields plus `owner`
a	$\in$	Val	::=	`this`, local values, value parameters, and singleton objects
x	$\in$	Var	::=	local variables
Γ	$\in$	$Environment$	::=	maps members of Val and Var to a Type T

Figure 3.1: Syntax and type environment.

Figure 3.1 summarizes the syntax and the naming conventions used. Like in previous Universe type systems, a type ($Type$) consists of two parts. In our case a standard Scala reference type ($SType$) is prefixed by an ownership modifier ($Univ$), which is either one of the `unknown` or `any` modifiers or a Universe path. As usual the ownership modifier describes the context of an expression and thus restricts the set of compatible objects from all objects of the defined Scala type (and its subtypes) to a subset thereof.

Each Universe path ($UPath$) starts with a mandatory object path and may be followed by a sequence of contexts. The $Context$ sort includes the `peer` and `rep` modifiers known from previous

Universe type systems and in addition the `up`, `reps`, and `ups` modifiers introduced in Section 2.6.1.

Object paths ($OPath$) consist of a start value or variable which may be followed by a sequence of field selections. Just like for regular field selections, only fields defined by the class of the prefix or one of its superclasses can be selected. Additionally the $Fields$ sort includes the virtual `owner` field, referencing the objects owner. The start of an object path can be either a local variable (Var) or an element of the Val sort, which contains only immutable elements. Namely it contains the current `this` reference, all local values, any method parameters in scope, and the reachable singleton objects. Of course members of the Val, Var and $Field$ sorts need to be of reference type, meaning they have to conform to `scala.AnyRef`.

Finally there is the type environment mapping function Γ, which is explained in detail in section 3.1.3.

Naming conventions In the following we use the term *universe* to refer to ownership modifiers ($Univ$) and *context* for the set of all objects with the same owner. Besides that we refer to the subclass of path-dependent universes ($UPath$) with the term *path*. To avoid confusion the members of the $Context$ sort are referred to as *context modifiers*. And finally we use the `this` constant to identify the current instance of the encompassing class.

Instead of having two classes of fields to distinguish mutable and immutable ones, we use a static *mutable* property. If $f.mutable$ equals $true$, field f is mutable, otherwise it is not.

Notation $\overline{T}$ denotes a sequence of Ts and the i-th element is denoted by T_i. Such a sequence can be empty and the empty sequence is denoted by ϵ. We use subscripts to differ between several expressions of the same kind. Decomposition of expressions is then done implicitly and the subscripts are again used to help identifying the relations between the original and the decomposed expression. For example we implicitly decompose the type T_0 into its universe u_0 and its Scala type C_0. Decomposition of tuples is also done with subscripts or, for anonymous tuples, via projection $\downarrow_i$ to select the i-th component of a tuple. Indexing for sequences and tuples ranges from 1 to n with n being the cardinality of the sequence or tuple.

Since, as stated above, a type is a tuple $u\ C$ consisting of a universe u and a Scala type C, we write other occurrences of tuples as (x, y) to avoid confusion. Sequences of tuples, however, are denoted by $\overline{C\ x}$ which we believe to be better readable.

3.1.2 Classifying path types

$$
\begin{aligned}
&isStable :: Univ \rightarrow bool \\
&\quad isStable(a.\overline{f}) \quad = \quad \forall i :! f_i.mutable \\
\\
&isExact :: Univ \rightarrow bool \\
&\quad isExact(a.\overline{f}.c) \quad = \quad isStable(a.\overline{f}) \wedge c \neq \texttt{reps}, \texttt{ups}
\end{aligned}
$$

Figure 3.2: The $isStable$ and $isExact$ predicates.

The predicates in Figure 3.2 are used to distinguish the different classes of paths, as defined in Section 2.6.3. The most important class for typing purposes is formed by the stable paths. They are a core requirement for path-dependent types in any type system. We use the $isStable$ and $isExact$ predicates to test for stable and exact paths. Loose paths are then all paths which are not covered by either of these predicates.

3.1.3 Type mapping and path normalization

The type environment mapping function Γ, as depicted by Figure 3.3, maps variables to their declared type but values, including the `this` reference, to their declared Scala type annotated

$$
\begin{array}{l}
\Gamma :: \mathit{Start} \rightarrow \mathit{Type} \\
\quad \Gamma(a) \;=\; a\ C \text{ with } C \text{ being the declared Scala type of } a \\
\quad \Gamma(x) \;=\; \text{declared } \mathit{Type}\ T \text{ of variable } x
\end{array}
$$

Figure 3.3: Environment mapping.

with the identifier of the value. Examples include *this Object* or *myLocalValue String*. Such types are needed by the type system in order to construct stable paths for dependent types.

Following good practice we defined some defaults to allow for shorter and better readable program code. We do so by defining the additional *SUniv* sort and the validation and normalization function *normalize* (Figure 3.4). *SUniv* is a generalization of *Univ* and *normalize* is a partial function, which maps *SUniv* to *Univ* by applying the default values where appropriate. It also takes care of simplifying the paths and eliminating mutable components. If *normalize* is not applicable to a user-specified universe, the universe is not valid and has to be rejected. In particular stable paths given by programmers are rejected, as explained in Section 2.6.6.

$$
\mathit{SUniv} \quad ::= \quad \epsilon \mid \mathit{Univ} \mid \overline{\mathit{Field}}.\overline{\mathit{Context}}
$$

$$
\begin{array}{lcl}
\mathit{normalize} :: \mathit{SUniv} \rightarrow \mathit{Univ} & & \\
\quad \mathit{normalize}(\texttt{unknown}) & = & \texttt{unknown} \\
\quad \mathit{normalize}(\texttt{any}) & = & \texttt{any} \\
\quad \mathit{normalize}(\epsilon) & = & \texttt{this.peer} \\
\quad \mathit{normalize}(\overline{f}.c_0.\overline{c}) & = & \mathit{normalize}(\texttt{this}.\overline{f}.c_0.\overline{c}) \\
\quad \mathit{normalize}(s.\overline{f}.c_0.\overline{c}) & = & \begin{cases}
\mathit{simplify}(s.\overline{f}.c_0.\overline{c}) & \text{if } \mathit{isStable}(s.\overline{f}) \\
\mathit{normalize}(\mathit{simplify}(s.\overline{f}.c_0.\overline{c})) & \\
\qquad \text{if } \mathit{simplify}(s.\overline{f}.c_0.\overline{c}) \neq s.\overline{f}.c_0.\overline{c} & \\
\mathit{normalize}(\mathit{simplify}(\mathit{loosen}(s.\overline{f}.c_0.\overline{c}))) & \text{otherwise}
\end{cases}
\end{array}
$$

Figure 3.4: Path normalization

The class of valid user specified universes, as defined by *normalize*, contains the `unknown` and `any` modifiers as well as the empty modifier, which is expanded to the default value `this.peer`. Furthermore we allow some form of degenerate paths, namely paths missing a start value or variable and assume the start to be the `this` reference. This follows the typing rules for regular field selections in Scala, where the explicit specification of the leading `this` is optional as well. Paths that start with a stable prefix and end with a context modifier are valid and can be accepted after simplifying them. This eliminates path aliasing as discussed in Section 2.6.7. The next case tries to apply this lossless simplification step also to paths containing mutable components.

Finally, to eliminate remaining mutable components, all paths whose prefix is not stable and can not be simplified anymore are normalized by applying the *simplify* and *loosen* functions explained in Sections 3.1.6 and 3.1.7. This step eliminates mutable components, but since the resulting path represents a loosened form of the initial path, it should be communicated to the programmer by a warning.

3.1.4 Field selections

The major part of typing field selections is done by the *select* function in Figure 3.5. Its arguments are the type of the receiver expression, the identifier of the accessed field and the declared type of the field. The return value is a pair consisting of the combined universe for the selection and a boolean flag which indicates if type information got preserved during the selection or not.

Please note that *select* always returns a valid universe, since field reads should always be possible. On the other hand we can only permit field updates if the combined universe describes

the exact same context or set of contexts as the fields universe from the viewpoint of the receiver. The only difference should be the change of viewpoint. If any information is lost during the viewpoint adaptation, the resulting universe describes a bigger set of contexts and the field selection expression cannot be used as the left-hand side in an assignment, since a field update using this type would not be save. The boolean part of the result is therefore only needed for field updates, while field reads can simply discard it.

$$
\begin{array}{l}
select :: Type \times Field \times Type \to Type \times bool \\
\quad select(T_0, f, T_f) \quad = \quad \begin{cases} (u_0.f\ C_f, false) & \text{if } isStable(u_0) \wedge\, !f.mutable \\ subst(T_f, (T_0, \texttt{this})) & \text{otherwise} \end{cases} \\
\qquad \text{where } T_f \text{ is the declared } Type \text{ of } f \text{ and} \\
\qquad f \text{ is a field selection on some expression of } Type\ T_0
\end{array}
$$

Figure 3.5: Field selection.

The first case handled in *select* covers selections of immutable fields on receivers whose universe, as mapped by Γ, is a stable path. The resulting combined path is simply the path constructed by the concatenation of the receivers path and the field name, which is a stable path again. And since the field is immutable, assignments to it are impossible by definition. In any other case the resulting universe has to be computed using a viewpoint adaptation.

$$
\begin{array}{l}
subst :: Type \times \overline{Type \times Val} \to Type \times bool \\
\quad subst(u\ C, \overline{T_x\ x}) \quad = \quad (u_{res}\ C, b_{res}) \\
\qquad \text{where } (u_{res}, b_{res}) = subst1(u, \overline{u_x\ x}) \\
\\
subst1 :: Univ \times \overline{Univ \times Val} \to Univ \times bool \\
\quad subst1(\texttt{unknown}, \_) \quad = \quad (\texttt{unknown}, true) \\
\quad subst1(\texttt{any}, \_) \quad = \quad (\texttt{any}, true) \\
\quad subst1(s.\overline{f}.\overline{c}, \overline{u\ x}) \quad = \quad res \\
\qquad !\exists (u_i, x_i) : s = x_i : \quad res = (simplify(s.\overline{f}.\overline{c}), true) \\
\qquad \exists_1 (u_i, x_i) : s = x_i : \\
\qquad\quad u_i = \texttt{unknown} : \qquad res = (\texttt{unknown}, false) \\
\qquad\quad u_i = \texttt{any} : \qquad res = (\texttt{unknown}, false) \\
\qquad\quad isStable(u_i) : \qquad res = (simplify(u_i.\overline{f}.\overline{c}), true) \\
\qquad\quad !isStable(u_i) \wedge \left|\overline{f}\right| = 0 : \quad res = (simplify(u_i.\overline{c}), assignable(u_i, simplify(\texttt{this}.\overline{c}))) \\
\qquad\quad \text{otherwise} : \qquad res = (subst1(simplify(loosen(s.\overline{f}.\overline{c})), \overline{u\ x}) \downarrow_1, false)
\end{array}
$$

Figure 3.6: Viewpoint adaptation.

Viewpoint adaptation Viewpoint adaptations are used at two places in our type system, one being field selections and the other method invocations. We use the *subst* function to adapt a type with regards to a different viewpoint defined in terms of another type. Its arguments are the type to adapt and a sequence of type-identifier pairs defining different viewpoints. The second parameter needs to be a sequence, since we are using the same function to perform the viewpoint adaptations needed to type method invocations. For simple field selections that would not be necessary. Also while viewpoints and therefore also viewpoint adaptations are defined in terms of universes, we use types instead to allow for an easy extension of the type system to parameterized types, which is discussed in Section 3.2. To allow for better understanding we have moved the viewpoint adaptation into the auxiliary function *subst*1, which, for the non-parameterized case, is simply called by *subst*. The *subst* function of the parameterized system is slightly more complex but also makes use of *subst*1 to perform the actual viewpoint adaptation of a universe with regards to another universe.

For field selections[1] *subst* is invoked by *select* to substitute the `this` prefix of the fields path by the full universe of the receiver expression. If the field universe is `unknown` or `any`, the resulting universe trivially equals the field universe and assignments are allowed. For the case of the field universe being `unknown`, this might be confusing. Recall that we are not allowing the `unknown` modifier to be used as main modifier for fields. So allowing assignments will not break type safety here, since there will never be such an assignment. We need this behaviour of *subst*1 for the extension to parameterized types discussed in Section 3.2.

If the field universe is neither `any` nor `unknown`, it is a path and we try to match the start of it against one of the values in the viewpoint sequence supplied as second argument. If none matches, the result is simply the field path and as nothing has been substituted, no type information got lost and assignments can be allowed. This handles the case where the type of the field does not depend on the receiver but on some singleton object, in which case no viewpoint adaptation is necessary.

If the sequence contains a matching value x_i, a viewpoint adaptation can be performed and the result depends on the universe u_i paired with x_i. If u_i is `unknown` or `any`, the substitution result is `unknown`, and as this kind of viewpoint adaptation completely discards the field path, assignments are not safe and cannot be permitted. Otherwise if u_i is stable we can perform the viewpoint adaptation by simply replacing the start value s by the universe u_i and return the simplified result. For this case it is safe to allow assignments.

The next case then matches if u_i is not stable and the field path contains no field selections but only a start value plus a sequence of context modifiers. It returns the simplified composition of u_i with the context modifier sequence $\overline{c}$ from the field path. Whether assignments are permissible for this case is determined by a call to *assignable*. This helper function essentially checks whether it is possible to describe the exact same context after the viewpoint adaptation or whether this is impossible. This check helps us to determine whether assignments to a field through a mutable receiver can be allowed and it also helps us check whether a method can be called on a mutable receiver. Note that instead of using s as the start value for the second argument to *assignable*, we are using the `this` constant. This is needed for the viewpoint adaptation of method parameters as explained in the next subsection. For field selections s always equals `this` when this case is reached and it therefore makes no difference.

Finally in any other case the field path still contains field selections and the receiver expression is not stable. Combining these would result in an illegal path. To prevent this the field path is loosened[2] and the viewpoint adaptation calculated based on the resulting path. Since this operation loses type information, a path computed in this manner cannot be used to type field write expressions and the boolean part of the return value is set to false.

$$
\begin{array}{l}
assignable :: Univ \times Univ \rightarrow bool \\
\quad assignable(p.\texttt{reps}.\overline{c}, \_\_) \quad = \quad false \\
\quad assignable(p.\texttt{ups}.\overline{c}, \_\_) \quad = \quad false \\
\quad assignable(s.\overline{f}.\overline{c}, \texttt{this.peer.}\overline{\texttt{rep}}.\overline{\texttt{reps}}) \quad = \quad isStable(s.\overline{f}) \wedge (|\overline{c}| = 1 \vee \forall i : c_i = \texttt{up}) \\
\quad assignable(s.\overline{f}.\overline{c}.\overline{c_x}, \texttt{this.up.}\overline{c_y}.\overline{\texttt{rep}}.\overline{\texttt{reps}}) \quad = \quad isStable(s.\overline{f}) \wedge (|\overline{c}| = 1 \vee \forall i : c_i = \texttt{up}) \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \forall i : c_{xi} = \texttt{rep} \wedge |\overline{c_y}| + 1 \geq |\overline{c_x}| \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \forall j : (c_{yj} = \texttt{up} \vee c_{yj} = \texttt{ups})
\end{array}
$$

Figure 3.7: The `assignable` predicate.

Assignable The first argument to *assignable* (Figure 3.7) represents the receivers universe and the second the field universe. The first two cases forbid assignments to fields on receivers with a `reps` or `ups` modifier in the path, since in this case the context of the receiver is not exactly determinable. The remaining two cases are needed to check assignments if the receiver path is

[1] Method invocations are explained in Section 3.1.5.

[2] The function *loosen* used to achieve this is explained in detail in Section 3.1.6.

not stable, but the combined universe of receiver and field, as returned by *subst*, does not loose any information. This is actually a very common case and can occur for example when the receiver expression is a local variable. We are not going into details here, let us just state that the presented patterns cover most cases of permissible field updates on mutable receivers. A more detailed reasoning is given in Appendix A.

Note that the overbar over `reps` for the last two cases expresses that it is optional. We are, like for most other predicates and functions, assuming that the input paths are normalized. So according to the normalization and simplification function *simplify* (Section 3.1.7) a `reps` modifier may only occur once. The same holds for the `ups` modifier which may occur in the field path for the last case.

3.1.5 Checking method calls

Method calls can be seen and checked as a combination of at most one field read (for the return value) and zero or more field assignments (for the argument values). Because we allow method parameter types and the return type to not only depend on the receiver but also on other parameters, method calls are slightly more complex. First off, to prevent circular dependencies, we impose the restriction that a parameter may only depend on parameters preceding it in the method signature. The return value, however, may depend on any parameter. Now while this prohibits circular dependencies, it is still possible to have a dependency cascade. For example if, in a method with multiple parameters, every parameter except the first depends on its immediate predecessor.

To handle this, the previously defined *subst* function takes a sequence of type-value pairs. In order to type check a method invocation, *subst* has to be called once for each parameter and fed with the current parameters type and the sequence of all type-identifier pairs of preceding parameters augmented by the receivers type as replacement for `this`. The idea behind this is to treat the method parameters like virtual fields and simply check if assignments would be permissible for each of these virtual fields. Therefore the boolean part for all results of these *subst* invocations has to be *true*. Otherwise the method cannot be called since the viewpoint adaptation was lossy for some parameter and it is impossible to match this parameters context, as seen from the viewpoint of the caller, with an actual argument. For the return value it is fine if the viewpoint adaptation is lossy, since it is equivalent to a field read, not a field write.

```
class C {
  val x: this.rep Object = new this.rep Object

  def foo (a: this.peer C, b: a.rep Object) = {
    // ...
  }
}

val c: this.rep C = new this.rep C
val d: this.rep C = new this.rep C
var z: this.rep C = new this.rep C

c.foo(d, d.x)
c.foo(z, z.x)    // compile time error
```

Figure 3.8: Example of a method where one parameter depends on another parameter.

For an illustration of the method parameter handling consider the example in Figure 3.8 where parameter `b` of method `foo` is depending on parameter `a`. For an actual invocation of method `foo` it has to be checked whether the first argument is `peer` to the receiver of the method invocation and the second argument is `rep` to the first. In order to check the later we are treating parameter `b`

as a virtual field of parameter `a`, which again is handled as a virtual field of the receiver expression since its path starts with a leading `this`. The viewpoint adaptation performed by *subst* will then replace the `this` in `this.peer` by the universe of the actual receiver expression and the `a` in `a.rep` by the universe of the actual argument corresponding to parameter `a`.

Consider the invocation of method `foo` on line 12 where both, the receiver and the actual argument for parameter `a`, are values and therefore have a stable universe. The invocations of *subst* for this call are as follows:

- for parameter `a`: $subst(\texttt{this.peer C}, (\texttt{c C}, \texttt{this})) = (\texttt{this.rep C}, true)$
- for parameter `b`: $subst(\texttt{a.rep Object}, (\texttt{d C}, \texttt{a}), (\texttt{c C}, \texttt{this})) = (\texttt{d.rep Object}, true)$

The expected universe for argument `a` is computed to be `c.peer` which is simplified to `this.rep` and the expected universe for argument `b` is `d.rep`. Both of them are computed without loss of information and therefore qualify for assignments. Since the actual arguments are subtypes of their respective expected type, it is thus safe to allow this method call.

Now consider the second invocation of `foo` on line 13. For this invocation only the receiver has a stable universe. The invocations of *subst* for this call are as follows:

- for parameter `a`: $subst(\texttt{this.peer C}, (\texttt{c C}, \texttt{this})) = (\texttt{this.rep C}, true)$
- for parameter `b`:
  $subst(\texttt{a.rep Object}, (\texttt{this.rep C}, \texttt{a}), (\texttt{c C}, \texttt{this})) = (\texttt{this.rep.rep Object}, false)$

Since the universe of parameter `a` is depending only on the receiver, its expected argument type is the same as for the previous invocation. The expected universe for argument `b` on the other hand is computed to be `this.rep.rep` which is not exact enough to allow this method call. The loss of precision is detected by the *assignable* predicate. It correctly denies the possibility of matching the context for parameter `b` from the viewpoint of the caller. In order to do this it is called with the universe of the actual argument of `a` as receiver universe and `this.rep`, the universe of parameter `b` with start value `a` replaced by `this` as field universe: $assignable(\texttt{this.rep}, \texttt{this.rep}) = false$. Replacing `a` by `this` in `b`'s path is the crucial step in pretending `b` is a virtual field of `a`, respectively the argument for parameter `a`. Intuitively it is not possible to identify the exact context of a `rep` field if the receiver object, which `this.rep` is relative to, is not uniquely identified. A workaround which allows to perform the call to `foo` with these arguments would be to introduce a temporary local value referring to the same object as `z`.

3.1.6 Computing loose paths

$$loosen :: UPath \rightarrow Univ$$

$$loosen(p.f.\overline{c}) = \begin{cases} u.\overline{c} & \text{if } u \neq \texttt{unknown}, \texttt{any} \\ u & \text{otherwise if } |\overline{c}| = 0 \\ \texttt{unknown} & \text{otherwise} \end{cases}$$

where u_f is the universe declared for field f
and $u = subst1(u_f, (p, \texttt{this}))\downarrow_1$

$$loosen(s.\overline{c}) = \begin{cases} u_s.\overline{c} & \text{if } u_s \neq \texttt{unknown}, \texttt{any} \\ u_s & \text{otherwise if } |\overline{c}| = 0 \\ \texttt{unknown} & \text{otherwise} \end{cases}$$

where u_s is the universe declared for local variable or value s

Figure 3.9: Computing loose paths.

Figure 3.9 presents the function *loosen*, which essentially substitutes the last field selection by the declared universe of the field. The first rule computes this substitution and the second replaces

the start value by its declared universe. *loosen* is a lossy operation and every universe computed using *loosen* is a "supertype" of the initial path, since it describes a family of contexts including the originally described one. See Section 3.1.8 for more information about subtyping on paths.

Note that even though the composition of a universe with a sequence of contexts might result in a path that could be further simplified we are not doing this here. The reason behind this is that the path simplification function *simplify* uses *loosen* to eliminate path aliasing and in order to avoid infinite loops due to these functions mutually calling each other, we are not simplifying the composition here. We are instead simplifying the result of each use of *loosen* outside *simplify* at the call site.

3.1.7 Path simplification

$$
\begin{aligned}
&simplify :: Univ \to Univ \\
&\quad simplify(u) \quad = \quad \begin{cases} simplify(simplify1(u)) & \text{if } simplify1 \text{ is defined on } u \\ simplify(simplify2(u)) & \text{otherwise if } simplify2 \text{ is defined on } u \\ u & \text{otherwise} \end{cases} \\
&\\
&simplify1 :: UPath \to UPath \\
&\quad simplify1(p.\texttt{peer}.\texttt{peer}.\overline{c}) \quad = \quad p.\texttt{peer}.\overline{c} \\
&\quad simplify1(p.\texttt{peer}.\texttt{up}.\overline{c}) \quad = \quad p.\texttt{up}.\overline{c} \\
&\quad simplify1(p.\texttt{peer}.\texttt{ups}.\overline{c}) \quad = \quad p.\texttt{ups}.\overline{c} \\
&\quad simplify1(p.\texttt{rep}.\texttt{peer}.\overline{c}) \quad = \quad p.\texttt{rep}.\overline{c} \\
&\quad simplify1(p.\texttt{rep}.\texttt{up}.\overline{c}) \quad = \quad p.\texttt{peer}.\overline{c} \\
&\quad simplify1(id.\texttt{owner}.\texttt{peer}.\overline{c}) \quad = \quad id.\texttt{up}.\overline{c} \\
&\quad simplify1(id.\texttt{owner}.\texttt{rep}.\overline{c}) \quad = \quad id.\texttt{peer}.\overline{c} \\
&\quad simplify1(p.\texttt{up}.\texttt{peer}.\overline{c}) \quad = \quad p.\texttt{up}.\overline{c} \\
&\quad simplify1(p.\texttt{ups}.\texttt{peer}.\overline{c}) \quad = \quad p.\texttt{ups}.\overline{c} \\
&\quad simplify1(p.\texttt{reps}.\texttt{peer}.\overline{c}) \quad = \quad p.\texttt{reps}.\overline{c} \\
&\\
&\quad simplify1(p.\texttt{ups}.\texttt{up}.\overline{c}) \quad = \quad p.\texttt{up}.\texttt{ups}.\overline{c} \\
&\quad simplify1(p.\texttt{reps}.\texttt{rep}.\overline{c}) \quad = \quad p.\texttt{rep}.\texttt{reps}.\overline{c} \\
&\\
&\quad simplify1(p.\texttt{ups}.\texttt{ups}.\overline{c}) \quad = \quad p.\texttt{up}.\texttt{ups}.\overline{c} \\
&\quad simplify1(p.\texttt{reps}.\texttt{reps}.\overline{c}) \quad = \quad p.\texttt{rep}.\texttt{reps}.\overline{c} \\
&\\
&simplify2 :: UPath \to UPath \\
&\quad simplify2(p) \quad = \quad simplify1(loosen(p))
\end{aligned}
$$

Figure 3.10: Simplifying loose paths.

The *simplify* function depicted in Figure 3.10 is used to normalize sequences of context modifiers and is defined in terms of two helper functions. The first, *simplify*1, simplifies sequences of context modifiers where the first block of simplifications mostly takes advantage of the fact that the `peer` modifier stands for "the same context as". This means that a `peer` modifier in a path does denote the context identified by its path prefix. The following explanations are assuming that p equals `this` and no additional modifiers are given, i.e. $|\overline{c}| = 0$. The extension to other, more elaborate, p and additional context modifiers is straightforward. For better understanding of the simplifications take a look at Figures 2.9 and 2.10 in Section 2.6.1 where the meaning of the context modifiers and the virtual `owner` field are illustrated.

More than one `peer` modifiers in a row can, for obvious reasons, be replaced by just one. Furthermore since `up` designates the context containing the owner of the current object and because the current objects peers are owned by the same object, the `peer` modifier is redundant for the second case as well. The same holds for the third case. Also objects which are `peer` to a `rep`

object are simply `rep` themselves. The context encompassing a `rep` object is the current context `peer`. Addressing the peers of the current objects owner can be done by simply using `up`. The objects owned by the current objects owner are the current object and its peers. The `up` reference already includes all objects in the context of the current objects owner. And finally a trailing `peer` modifier does not change the contexts identified by an `ups` or `reps` modifier.

The second block of rules takes care of the ordering and the third normalizes sequences of `ups` and `reps`. These normalization and ordering rules help maintaining compatibility of equivalent types and keeping the type system tidy.

$simplify2$, the second helper function, tries to apply $simplify1$ to its loosened argument. This reduces the aliasing problems induced by path-dependent universes as explained in Section 2.6.7, and thus forms an important part of our system. If $simplify1$ is defined for the loosened path, the loosened path component was redundant and the application of *loosen* and $simplify1$ returns a path equivalent to p. If it is not defined, $simplify2$ is not defined for p and nothing will happen.

3.1.8 Subtyping

$$<: :: Type \times Type \rightarrow bool$$
$$u_0\ C_0 <: u_1\ C_1 \quad \Leftrightarrow \quad C_0 <:_{SC} C_1 \wedge u_0 <:_u u_1$$
where $<:_{SC}$ is the default subtype relation on Scala types

$$<:_u :: Univ \times Univ \rightarrow bool$$

`unknown`	$<:_u$	`any`	
u	$<:_u$	`unknown`	if $u \neq$ `any`
p.`rep`	$<:_u$	p.`reps`	
p.`rep.reps`	$<:_u$	p.`reps`	
p.`up`	$<:_u$	p.`ups`	
p.`up.ups`	$<:_u$	p.`ups`	
id	$<:_u$	id.`peer`	
id.`owner`	$<:_u$	id.`up`	
u	$<:_u$	$simplify(loosen(u))$	
$simplify(u)$	$<:_u$	u	
u	$<:_u$	u''	if $u <:_u u' \wedge u' <:_u u''$
$p.\bar{c}$	$<:_u$	$simplify(p'.\bar{c})$	if $p <:_u p'$

Figure 3.11: Subtyping.

A Universe type is a subtype ($<:$) of another Universe type if the respective Scala types are subtypes in the sense of the default Scala subtype relation $<:_{SC}$, and the ownership modifiers are subtypes as defined by the subtype relation on universes $<:_u$. For the universe subtype relation, `unknown` is a subtype of `any` and any universe, except `any`, is a subtype of `unknown`. Sequences of `rep` modifiers can be covered by the more general `reps` modifier. The same holds for sequences of `up` modifiers and `ups`. Some stable path id is a specialization and therefore a subtype of id.`peer`. Along the same lines id.`owner` is a subtype of the more general id.`up`. Furthermore any universe u is a subtype of the universe computed by applying *loosen* and $simplify$ to it. An example type hierarchy illustrating this rule can be seen in Section 2.6.8.

Subtyping on universes is of course reflexive and transitive. Where reflexivity is contained in the rule that $simplify(u)$ is a subtype of u. And finally if two paths are subtypes, the paths generated by selecting the same sequence of context modifiers on them are subtypes as well. A straightforward example of this last rule would be that from `this` $<:_u$ `this.peer` follows that `this.rep` $<:_u$ `this.peer.rep`. Along the same lines it can also be shown that id.`peer` is a subtype of id.`up.rep`. Since id.`owner` $<:_u$ id.`up` one can conclude that id.`owner.rep` $<:_u$ id.`up.rep` and by using that $simplify(u) <:_u u$ it follows that id.`peer` $<:_u$ id.`owner.rep` and therefore also id.`peer` $<:_u$ id.`up.rep`.

3.1.9 Type rules

The type rules can be found in Figure 3.12. The judgement $\Gamma \vdash e : T$ expresses that expression e is well typed with type T in environment Γ.

$$Subsumption: \frac{\Gamma \vdash e_0 : T \quad T <: T'}{\Gamma \vdash e_0 : T'} \qquad New: \frac{isExact(u) \vee u = \texttt{any}}{\Gamma \vdash \texttt{new}\ T : T}$$

$$LocalRead: \frac{s \in dom(\Gamma)}{\Gamma \vdash s : \Gamma(s)} \qquad VarWrite: \frac{x \in dom(\Gamma) \quad \Gamma \vdash e : \Gamma(x)}{\Gamma \vdash x := e : \Gamma(x)}$$

$$FieldRead: \frac{\Gamma \vdash e_0 : T_0 \quad fType(C_0, f) = T_1 \quad select(T_0, f, T_1) = (T_{res}, \_\_)}{\Gamma \vdash e_0.f : T_{res}} \qquad FieldWrite: \frac{\Gamma \vdash e_0 : T_0 \quad fType(C_0, f) = T_1 \quad f.mutable \quad select(T_0, f, T_1) = (T_{res}, true) \quad \Gamma \vdash e_2 : T_{res}}{\Gamma \vdash e_0.f := e_2 : T_{res}}$$

$$Invoc: \frac{\begin{array}{c}\Gamma \vdash e_0 : T_0 \quad mType(C_0, m) = T_{ret}\, m(\overline{T_p\ x}) \quad \Gamma \vdash \overline{e_2} : \overline{T_2} \\ \forall i : subst(T_{pi}, (T_0, \texttt{this}), \overline{T_2\ x}) = (T_{ai}, true) \quad \forall i : T_{2i} <: T_{ai} \\ subst(T_{ret}, (T_0, \texttt{this}), \overline{T_2\ x}) = (T_{res}, \_\_)\end{array}}{\Gamma \vdash e_0.m(\overline{e_2}) : T_{res}}$$

Figure 3.12: Type rules.

Expressions typed with T can, according to the *Subsumption* rule, also be typed with T's supertypes. Object creation (*New*) is only allowed for exact paths and `any`, where creating an object with an `any` modifier can be used as a means of creating shared objects that are not part of a specific context. This is very useful when using objects that possibly expose their peers or mess with them, which can be prevented by putting such objects into an anonymous context.

Values and variables are typed by a lookup in the type environment (*LocalRead*). Assignments to variables (*VarWrite*) are straightforward. *FieldRead* uses the *select* function to determine the resulting path. Since it is not of interest to us, we do not give a definition of the *fType* function used to lookup the declared type of a field. Its meaning should be clear nevertheless. Field updates are covered by the *FieldWrite* rule. The resulting type is computed in the same way as for field reads. Additionally the field has to be mutable and the second part of *subst*s return value has to be *true*, otherwise type information got lost during the viewpoint adaptation and the update would break type safety. Finally method invocations (*Invoc*) can be typed like a sequence of field updates with the parameters being virtual fields updated on the receiver, as described in Section 3.1.5.

The rules for casts and the `null` reference are straightforward and would not provide any substantial contributions to the presented type system. They are therefore not listed.

Initialization of local values and immutable fields can be covered by handling them like an assignment to a local variable or a field write respectively.

3.2 Parameterized types

This section extends our system to parameterized types. In order to do this we need to extend the syntax and redefine the environment mapping function Γ as well as the *select* and *subst* functions

used for the viewpoint adaptations. Furthermore we also need to adapt the subtyping operator <:. All other functions are only concerned with universes and are therefore not influenced by the extension to parameterized types. Finally we do not need to update the type rules since they are already expressive enough to cover the extended system.

3.2.1 Syntax

$$
\begin{array}{lcll}
T & \in & Type & ::= \quad NType \mid TVarId \\
N & \in & NType & ::= \quad Univ\ SType[\overline{Type}] \\
C & \in & SType & ::= \quad \text{non-parameterized Scala type conforming to } \texttt{scala.AnyRef} \\
X & \in & TVarId & ::= \quad \text{type variable}
\end{array}
$$

Figure 3.13: Syntax extension for parameterized types.

Figure 3.13 extends the syntax of our type system to parameterized types. $SType$ is explicitly restricted to non-parameterized Scala types now and the newly introduced $NType$ sort then denotes non-variable types with ownership annotation. The type parameters for a parameterized type can again be non-variable types or type variables ($TVarId$). The $Type$ sort is therefore extended to subsume both, non-variable types and type variables.

Naming conventions We use again static properties to cover Scala's variance annotations for parameterized types. If $X.cov$ equals $true$, type variable X is declared covariant and if $X.cont$ equals $true$, it is declared contravariant. Otherwise if neither $X.cov$ nor $X.cont$ are $true$, the type variable is declared invariant. Of course at most one of $X.cov$ and $X.cont$ can be $true$ for a particular type variable.

3.2.2 Type mapping and auxiliary functions

$$
\begin{array}{l}
\Gamma :: Start \rightarrow Type \\
\quad \Gamma(a) \quad = \quad a\ T \text{ with } T \text{ being the declared type of value } a \\
\quad \Gamma(x) \quad = \quad \text{declared } Type\ T \text{ of variable } x \\
\quad \Gamma(X) \quad = \quad \text{upper bound } N \text{ of type variable } X
\end{array}
$$

Figure 3.14: The environment mapping function for parameterized types.

Figure 3.14 shows the adapted environment mapping function Γ. Values, including the `this` reference, are mapped to their declared $NType$ or type variable annotated with the identifier of the value and variables to their declared type. Finally type variables are mapped to their upper bound which defaults to `any scala.AnyRef`.

$$
\begin{array}{l}
setMain :: NType \times Univ \rightarrow NType \\
\quad setMain(u\ C[\overline{T}], u_0) \quad = \quad u_0\ C[\overline{T}]
\end{array}
$$

Figure 3.15: The auxiliary function $setMain$.

We use the auxiliary function $setMain$ depicted in Figure 3.15 to modify the main modifier of a parameterized type.

$$
\begin{array}{l}
select :: NType \times Field \times Type \rightarrow Type \times bool \\
\quad select(T_0, f, T_f) \quad = \quad res \\
\qquad\qquad isStable(u_0) \wedge !f.mutable : \quad res = (setMain(T_{tmp}, u_0.f), false) \\
\qquad\qquad \text{otherwise} : \qquad\qquad\qquad res = (T_{tmp}, b_{tmp}) \\
\qquad\qquad \text{where } (T_{tmp}, b_{tmp}) = subst(T_f, (T_0, \texttt{this})), \\
\qquad\qquad T_f \text{ is the declared } Type \text{ of } f, \\
\qquad\qquad \text{and } f \text{ is a field selection on some expression of } Type\ T_0
\end{array}
$$

Figure 3.16: Field selection for parameterized types

3.2.3 Field selections

Field selections in the presence of parameterized types always need a viewpoint adaptation to make sure all type arguments are updated to relate to the changed viewpoint. For the case of a stable receiver universe and the selected field being immutable, the main modifier of the viewpoint adapted field type is set to be the concatenation of the receiver universe and the field identifier. In any other case the result is simply the viewpoint adapted field type.

Note that the first parameter for *select* is required to be a non-variable type, since field selections require the universe of the receiver to be known.

3.2.4 Viewpoint adaptation

$$
\begin{array}{l}
subst :: Type \times \overline{Type \times Val} \rightarrow Type \times bool \\
\quad subst(X, (u_T\ C_T[\overline{T_T}], \texttt{this}), \overline{T\ x}) \quad = \quad \begin{cases} (T_{T_X}, true) & \text{if } T_{T_X} \downarrow_1 \neq \texttt{unknown} \\ (T_{T_X}, false) & \text{otherwise} \end{cases} \\
\qquad\qquad \text{where } X \in dom(C_T) \text{ and } T_{T_X} \text{ is at the corresponding position} \\
\quad subst(u_0\ C[\overline{T_0}], \overline{T\ x}) \qquad\qquad = \quad res = (u_1\ C[\overline{T_r}], b_r) \\
\qquad \text{where } (u_1, b_1) = subst1(u_0, \overline{u\ x}) \\
\qquad \forall i : (T_{2i}, b_{2i}) = subst(T_{0i}, \overline{T\ x}) \\
\qquad \forall i : (T_{ri}, b_{ri}) = \begin{cases} (T_{2i}, false) & \text{if } !b_{2i} \wedge X_i.cov \\ (setMain(T_{2i}, \texttt{unknown}), false) & \text{if } !b_{2i} \wedge !X_i.cov \\ (T_{2i}, true) & \text{otherwise} \end{cases} \\
\qquad\qquad \text{where } X_i \text{ is the type variable at position } i \\
\qquad b_r = b_1 \wedge \bigwedge_{\forall i} b_{ri}
\end{array}
$$

Figure 3.17: Viewpoint adaptation for parameterized types.

The extended *subst* function shown in Figure 3.17 performs the adaptations of possibly parameterized types with regards to another type. The first rule is responsible for replacing occurrences of type variables by the actual type arguments at the corresponding position. If the main modifier of the replacement is `unknown`, assignments can not be allowed since the `unknown` modifier encodes a specific yet `unknown` context that is impossible to match statically. Note that if type variable X is not in the domain of C_T this is a violation of type well-formedness and will be forbidden by the well-formedness checks to be developed in future work.

The viewpoint adaptation for parameterized types is performed by first adapting the main modifier using the previously defined function *subst*1 and then recursively adapting the type arguments. As a next step the main modifiers of type arguments which have lost information during the viewpoint adaptation are set to `unknown` unless the type variable at the corresponding position is defined to be covariant. In this case it is safe to keep the widened type. Finally the resulting type is assembled and the assignable flags for all type parameters and the main modifier are collected and passed on.

3.2.5 Subtyping

$$
\begin{array}{l}
<: :: Type \times Type \rightarrow bool \\
\quad u_0\ C[\overline{T_0}] \quad <: \quad u_1\ C[\overline{T_1}] \quad \Leftrightarrow \quad u_0 <:_u u_1 \wedge \forall i : (\\
\qquad\qquad\qquad\qquad\qquad\qquad\quad (X_i.cov \wedge T_{0i} <: T_{1i}) \vee \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad (X_i.cont \wedge T_{1i} <: T_{0i}) \vee \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad (T_{0i} <:_{arg} T_{1i})) \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{where } X_i \text{ is the type variable at position } i \\
\qquad X \quad <: \quad \Gamma(X) \\
\qquad X \quad <: \quad X
\end{array}
$$

Figure 3.18: Subtyping for types.

Two parameterized types are subtypes (<:) if their Scala classes are both the same, their main modifiers are subtypes and the type arguments are related according to the respective variance annotation. If a type variable is declared co- or contravariant, the actual type argument has to obey this relation. If it is declared invariant we can still allow a limited form of covariance similar to the one presented for Generic Universe Types [8]. In particular limited covariance for invariant type arguments ($<:_{arg}$) allows element types to be subtypes of their respective type with `unknown` as main modifier. To protect type safety, $<:_{arg}$ is defined recursively.

$$
\begin{array}{l}
<:_{arg} :: Type \times Type \rightarrow bool \\
\quad u\ C[\overline{T_0}] <:_{arg} \texttt{unknown}\ C[\overline{T_1}] \quad \Leftrightarrow \quad \forall i : T_{0i} <:_{arg} T_{1i} \\
\quad u\ C[\overline{T_0}] <:_{arg} u\ C[\overline{T_1}] \quad \Leftrightarrow \quad \forall i : T_{0i} <:_{arg} T_{1i}
\end{array}
$$

Figure 3.19: Limited covariant subtyping for invariant type arguments.

The subtyping relation for types also ensures that a type variable X is a subtype of itself and its upper bound $\Gamma(X)$.

3.3 Runtime Types for Scala

Polymorphism and the substitution principle make it impossible to statically type check everything. Therefore modern object-oriented languages provide a runtime environment which allows programmers to retain type safety while performing statically unsafe operations like for example downcasts.

We refer to the Java target platform for the considerations regarding runtime types. The extension to the .net platform is expected not to introduce any further requirements for Universe runtime types.

3.3.1 Runtime checks in Java

The Java Language Specification [26] lists five places, where the actual class of a referenced object affects program execution in a manner, that cannot be deduced from the static type of the expression:

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

3.3.2 Runtime checks in the Universe type system

The extended types of the Universe type system have no influence on the dynamic checks needed for method invocation and exception handling. Method invocation does not need to be handled, because the Universe types do not introduce different methods and therefore do not influence dynamic method binding. And exception handling does not change, since exceptions are best caught and forwarded using `any` references which do not restrict passing. When adhering to the owner-as-modifier discipline this means that exceptions are not modifiable. But we consider modification of exceptions to be bad practice anyway and suggest to avoid it in favor of wrapping.

More insight on exceptions in ownership type systems is found in [9].

The `instanceof` operator and casts of reference types A motivational example for having runtime checks for casts and the `instanceof` operator[3] would be the cast of an `unknown` or `any` reference to a `rep` reference by its owner. The typecase matches in Scala's pattern matching are another example. However, we claim that path-dependent types should reduce the need for downcasts by a considerable amount. Together with parameterized types it should be possible to almost always avoid the use of such statically unsafe operations.

Note also that the checks are only necessary if the ownership modifier changes. If only the Scala class changes the universe check would be superficial.

Assignment to an array component of reference type The Java Language Specification states that type-checking rules allow the array type `T[]` to be treated as a subtype of `S[]` if `T` is a subtype of `S`, but this requires a run-time check for assignment to an array component, similar to the check performed for a cast. Otherwise it would be possible to use a reference of type `S[]` to put an object of type `S` into an array of type `T[]` which would obviously break type safety.

While arrays in Scala are not covariant it is still possible to explicitly upcast a reference of type `Array[T]` to `Array[S]` if `T <: S`. We do, however, think that this should be avoided and have no intentions of supporting this with regards to Universe types. We therefore statically forbid such casts of array types if the element modifier changes.

3.3.3 Runtime representation of path-dependent Universe types

In our understanding, path-dependent types are mainly a static feature. This means for the main modifiers we identified no need for special or adapted runtime types reflecting the paths at runtime. We even believe it is not feasible, since path-dependency is an attribute of the reference and not the object. Since the same object can be referred to via many different paths, it would be impractical to store this information in the runtime type. We believe it suffices to have a virtual `owner` field referencing an objects owner. This should also make it possible to compute the relation of any two objects at runtime if necessary.

For parameterized types we additionally need to store the path for the type arguments. This is necessary since storing only the owner is not expressive enough. For example for a collection of type `this.peer List[this.rep.rep Object]` the elements can have different owners, namely all objects owned by `this`.

3.3.4 Proposed implementation for the Universe runtime checks

In the following we briefly sketch how the runtime checks can be implemented. We are basing our considerations on our earlier work on runtime checks for Universe Java.[25]

Object creation The Universe runtime checks for Java used a global hash table to register all objects, at their point of creation, and store the ownership information in the form of a reference to the owner object. Since Java has been released under the terms of the General Public Licence

[3] Scala uses the predefined functions `isInstanceOf` and `asInstanceOf` to perform type checks and casts.

(GNU GPL) in the meantime, a modification of `java.lang.Object`, to add support for the owner field, or the Java virtual machine (JVM), might be considerable. With regards to performance, such a solution would be preferable but all of these options have to be carefully evaluated for an actual implementation of the Universe runtime checks for Scala.

Casts (`asInstanceOf`) and `isInstanceOf` The basic idea for casts and `isInstanceOf` is to compare the stable prefix of the target path with the object being checked or casted. In the example depicted by Figure 3.20 the check involves a comparison of the owner of value `a` with the current `this` reference.

```
val a: any C = new this.rep C
// ...

val c: this.rep C =
    a.asInstanceof[this.rep C]
```

```
val a: any C = new this.rep C
// ...
if ( a.owner != this )
  throw new ClassCastException
val c: this.rep C =
    a.asInstanceof[this.rep C]
```

Figure 3.20: Sketch of the general idea for the implementation of runtime checks for Scala.

If the modifier of `c` were `peer`, the check would simply compare both owners (i.e. `a.owner == this.owner`) and if it were `up`, `a.owner` would be compared to `this.owner.owner`. Additional modifiers and combinations of modifiers can be covered along the same lines and are therefore not discussed here.

3.4 Additional restrictions to enforce encapsulation

In order to enforce the owner-as-modifier discipline again, we only need to adjust the type rules for object creation, assignments to fields, and method invocations. Generally it can only be permitted to modify an object if it is reachable through a `this.peer` or `this.rep` reference.

$$\begin{aligned} &isModifiable :: Univ \rightarrow bool \\ &\quad isModifiable(\texttt{this}.c) \quad = \quad (c = \texttt{peer}) \vee (c = \texttt{rep}) \end{aligned}$$

Figure 3.21: The *isModifiable* predicate defines the universes that qualify a reference for modification when adhering to the owner-as-modifier discipline.

The adapted rules enforce this for assignments to fields and invocation of impure methods by restricting the receiver using the *isModifiable* predicate from Figure 3.21. Creation of objects has also to be restricted in order to prevent an object from tainting other contexts by injecting objects.

Note that for pure methods, i.e. side-effect free methods, the universe of the receiver is not restricted. We are again using a static boolean property to indicate side-effect free methods. If $m.pure$ equals *true*, method m is side-effect free.

$$New: \frac{isModifiable(u) \vee u = \texttt{any}}{\Gamma \vdash \texttt{new}\ T : T} \qquad FieldWrite: \frac{\begin{array}{c}\Gamma \vdash e_0 : T_0 \\ isModifiable(u_0) \\ fType(C_0, f) = T_1 \\ f.mutable \\ select(T_0, f, T_1) = (T_{res}, true) \\ \Gamma \vdash e_2 : T_{res}\end{array}}{\Gamma \vdash e_0.f := e_2 : T_{res}}$$

$$Invoc: \frac{\begin{array}{c}\Gamma \vdash e_0 : T_0 \\ isModifiable(u_0) \vee m.pure \\ mType(C_0, m) = T_{ret}\, m(\overline{T_p\ x}) \\ \Gamma \vdash \overline{e_2} : \overline{T_2} \\ \forall i : subst(T_{pi}, (T_0, \texttt{this}), \overline{T_2\ x}) = (T_{ai}, true) \\ \forall i : T_{2i} <: T_{ai} \\ subst(T_{ret}, (T_0, \texttt{this}), \overline{T_2\ x}) = (T_{res}, \_\_)\end{array}}{\Gamma \vdash e_0.m(\overline{e_2}) : T_{res}}$$

Figure 3.22: Adapted type rules enforcing the owner-as-modifier discipline.

Chapter 4

Discussion

In this chapter we are discussing the type system developed in Section 3. We also present a prototype implementation of the type checker and use it to visualize the typing process for a number of examples.

4.1 Prototype implementation

To test our design and experiment with different options for some of the problems we had to solve, we decided to implement a prototype version of the type checker. Of course we used Scala to do this and it turned out that the functional elements in Scala made it very easy to transfer the functions and predicates of the formalization to actual Scala code. Especially pattern matching was very helpful. And thanks to object-orientation we could easily map the sorts our syntax defines to classes, subclasses and singleton objects. For example it enabled us to define the predicates and functions as methods of the class of their respective first parameter. Polymorphism and dynamic method binding then helped us to define the functionality at the appropriate place. In other words we used dynamic method binding to do the matching on the first parameter and pattern matching for the rest. For example the *subst*1 cases with the first argument being `unknown` or `any` are defined in the `Univ` class and the cases involving paths in the more specific `UPath` class that extends `Univ`.

We used different classes depending on the number of type arguments to represent parameterized types, which makes sure all type arguments are specified upon instantiation. We also use this technique for methods. Supplying a wrong number of arguments will therefore raise a compile time error. And of course the implementation prints the warning messages we suggested in Section 2.6.5 in case the normalization of a path is lossy.

The source code of the prototype implementation can be found in Appendix C. The implementation consists of four main files: `universes.scala` contains the classes from Figure 3.1 and the functions defined on universes. The classes from Figure 3.13 and the functions defined on types are found in `types.scala`. The other two files contain helper functions and classes.

4.1.1 Notation

For loose paths with no field selections we could actually have used Scala's implicit coercion functions to get a notation identical to the one used in the report. Using the same notation for all paths was, however, a core requirement for the prototype implementation. We therefore decided to use infix methods to mimic the look and feel of the default notation. See Figure 4.1 for an example.

```
class Data
class List[+X]

val l: this.rep List[any Data]
```

Notation used in the report.

```
val Data = Class("Data")
val List = Class("List", "+X")

val l = This°Rep ## List(Any ## Data)
```

Notation used by the prototype implementation.

Figure 4.1: Example showing the notation used by the prototype implementation.

4.1.2 Experiences with Scala

During the work on the prototype implementation we learned to know the beauty of several other Scala features by practical experience. We were impressed by the power given to the language by several small constructs. For example implicit definitions are an incredible tool, allowing programmers, amongst other things, to add methods to existing classes. Also pattern matching [11] proved to be incredibly helpful. Generally the concepts borrowed from functional languages, like higher-order functions, function passing, and anonymous functions, make it a joy to write Scala code. And of course there are several other comfort features like the local type inference [22], the infix notation for methods, and last but not least the freedom of not having to end each line with a semicolon.

4.1.3 Test suite

Along with the prototype implementation we have also developed a test suite to ensure the semantics of the prototype implementation were not altered during refactorings and adjustments.

4.2 Examples

This section presents some examples and investigates the typing process using the prototype implementation. In order to ensure the examples compile with the standard Scala compiler, we were using inline comments for the Universe annotations.

4.2.1 Field selection

```
class D {
    var g = new this.rep Object
}

class C {
    var f = new this.peer D
}

object client extends Application {
    var c = new this.peer C

    c.f = new this.peer D
    c.f.g = new this.peer.rep Object    // compile time error
}
```

Figure 4.2: A field selection.

Figure 4.3 shows the typing of a nested field selection expression `c.f.g`. The Scala code corresponding to this example is depicted in Figure 4.2. The viewpoint adapted type of expression `c.f` is computed to be `this.peer` (see the first typing block) and the resulting type for the expression `c.f.g` is `this.peer.rep` and assignments are not allowed, since *assignable* correctly forbids assignments to `rep` fields on mutable receivers (as can be seen in the second block).

```
select(this.peer C, f, this.peer D) = res
 |  calculating (Ttmp, btmp) ...
 |    subst(this.peer D, {(this.peer C,this)}) = res
 |     |  calculating (u1, b1) ...
 |     |    subst1(this.peer, {(this.peer,this)}) = res
 |     |     |    assignable(this.peer, this.peer) = true
 |     |     |res = (this.peer,true)
 |     |  (u1, b1) = (this.peer,true)
 |     |res = (this.peer D,true)
 |  (Ttmp, btmp) = (this.peer D,true)
 |res = (this.peer D,true)

select(this.peer D, g, this.rep Object) = res
 |  calculating (Ttmp, btmp) ...
 |    subst(this.rep Object, {(this.peer D,this)}) = res
 |     |  calculating (u1, b1) ...
 |     |    subst1(this.rep, {(this.peer,this)}) = res
 |     |     |    assignable(this.peer, this.rep) = false
 |     |     |res = (this.peer.rep,false)
 |     |  (u1, b1) = (this.peer.rep,false)
 |     |res = (this.peer.rep Object,false)
 |  (Ttmp, btmp) = (this.peer.rep Object,false)
 |res = (this.peer.rep Object,false)
```

Figure 4.3: The typing process for the field selection in Figure 4.2.

4.2.2 Method calls

The example for a method with dependent parameters has already been discussed in Section 3.1.5. The typing process in Figure 4.5 shows that the assignable predicate is only used when the *subst* function is replacing the start value of a path with a path that is not stable. The first block in Figure 4.5 represents the computation of the expected types for the arguments of the first function call (line 13 in Figure 4.4) and the second block the second call (line 14).

```
0  class C {
       val x = new this.rep Object

       def foo (a: this.peer C, b: a.rep Object) = {
           // ...
5      }
   }

   object client extends Application {
       val c = new this.rep C
10     val d = new this.rep C
       var z = new this.rep C

       c.foo(d, d.x)
       c.foo(z, z.x)    // compile time error
15 }
```

Figure 4.4: The method example from Figure 3.8 where parameter `b` depends on parameter `a`.

```
subst(this.peer C, {(c C,this)}) = res
 |  calculating (u1, b1) ...
 |    subst1(this.peer, {(c,this)}) = res
 |     |res = (this.rep,true)
 |  (u1, b1) = (this.rep,true)
 |res = (this.rep C,true)
subst(a.rep Object, {(d C,a), (c C,this)}) = res
 |  calculating (u1, b1) ...
 |    subst1(a.rep, {(d,a), (c,this)}) = res
 |     |res = (d.rep,true)
 |  (u1, b1) = (d.rep,true)
 |res = (d.rep Object,true)

subst(this.peer C, {(c C,this)}) = res
 |  calculating (u1, b1) ...
 |    subst1(this.peer, {(c,this)}) = res
 |     |res = (this.rep,true)
 |  (u1, b1) = (this.rep,true)
 |res = (this.rep C,true)
subst(a.rep Object, {(this.rep C,a), (c C,this)}) = res
 |  calculating (u1, b1) ...
 |    subst1(a.rep, {(this.rep,a), (c,this)}) = res
 |     |    assignable(this.rep, this.rep) = false
 |     |res = (this.rep.rep,false)
 |  (u1, b1) = (this.rep.rep,false)
 |res = (this.rep.rep Object,false)
```

Figure 4.5: The typing process for the method example in Figure 4.4.

4.2.3 Parameterized types

```
0 class ID
  class Data
  class Iter[X] (start: X) {
      var current: X = start
  }
5 class Node[K, V] (k: K, v: V) {
      var key: K = _
      var value: V = _
  }
  class Map[K, V] (k: K, v: V) {
10    var node = new this.rep Node[K, V] (k, v)
      var iter = new this.peer Iter[this.rep Node[K, V ]] ( node)
  }

  object client extends Application {
15    var id = new this.rep ID
      var data: any Data = new this.rep Data
      var map = new this.rep Map[this.rep ID, any Data] (id, data)

      var i = map.iter
20    map.iter = i    // compile time error

      var n: any Node[this.rep ID, any Data] = map.node
      n = i.current
      map.node = n    // compile time error
25    i.current = n   // compile time error

      id = n.key
      data = n.value
      n.key = id        // compile time error
30    n.value = data    // compile time error
  }
```

Figure 4.6: `Map` example.

Figure 4.6 presents a simplified version of the map example from GUT [8]. The client uses a field selection instead of a method call to get a reference to `map`'s iterator and the `Iter` class is parameterized with only one type argument.

The typing of the field selection expression `map.iter` (line 19) is depicted by Figure 4.7. The *select* function (Figure 3.16) invokes *subst* (Figure 3.17) to adapt the type of the `iter` field to the viewpoint of the `client` object. The *subst* function then uses *subst*1 (Figure 3.6) to adapt the main modifier of the field (`peer`) with regards to the main modifier of `map` (`rep`). The resulting main modifier (u_1) is again `this.rep` and the boolean flag b_1, indicates that the adaptation was lossless.

The next step computes $\overline{T_2}$, the viewpoint adaptation of the type argument of the iterator (here `this.rep Node[K, V]`). The adaptation of the main modifier for the only type argument is computed along the same lines as before and yields `this.rep.rep`. Then the type variables `K` and `V` are replaced by their corresponding arguments. The intermediate type ($\overline{T_2}$) for the type argument of `map.iter` is `this.rep.rep Node[this.rep ID, any Data]`. Since the computation of the main modifier `this.rep.rep` was lossy, it is set to `unknown` and the final type ($\overline{T_{res}}$) for the field selection is `this.rep Iter[unknown Node[this.rep ID, any Data]]`. The type of this expression in GUT would be `any Iter[any Node[this.rep ID, any Data]]`. Notice the different main modifiers. While in GUT the main modifier has to be set to `any` to ensure type

```
select(this.rep Map[this.rep ID, any Data], iter,
 |     this.peer Iter[this.rep Node[K, V]]) = res
 |  calculating (Ttmp, btmp) ...
 |    subst(this.peer Iter[this.rep Node[K, V]],
 |     |    {(this.rep Map[this.rep ID, any Data],this)}) = res
 |     |  calculating (u1, b1) ...
 |     |    subst1(this.peer, {(this.rep,this)}) = res
 |     |     |    assignable(this.rep, this.peer) = true
 |     |     |res = (this.rep,true)
 |     |  (u1, b1) = (this.rep,true)
 |     |  calculating T2 ...
 |     |    subst(this.rep Node[K, V],
 |     |     |    {(this.rep Map[this.rep ID, any Data],this)}) = res
 |     |     |  calculating (u1, b1) ...
 |     |     |    subst1(this.rep, {(this.rep,this)}) = res
 |     |     |     |    assignable(this.rep, this.rep) = false
 |     |     |     |res = (this.rep.rep,false)
 |     |     |  (u1, b1) = (this.rep.rep,false)
 |     |     |  calculating T2 ...
 |     |     |    subst(K, {(this.rep Map[this.rep ID, any Data],this)}) =
 |     |     |       (this.rep ID,true)
 |     |     |    subst(V, {(this.rep Map[this.rep ID, any Data],this)}) =
 |     |     |       (any Data,true)
 |     |     |  T2 = {(this.rep ID,true), (any Data,true)}
 |     |     |  Tres = {(this.rep ID,true), (any Data,true)}
 |     |     |res = (this.rep.rep Node[this.rep ID, any Data],false)
 |     |  T2 = {(this.rep.rep Node[this.rep ID, any Data],false)}
 |     |  Tres = {(unknown Node[this.rep ID, any Data],false)}
 |     |res = (this.rep Iter[unknown Node[this.rep ID, any Data]],false)
 |  (Ttmp, btmp) = (this.rep Iter[unknown Node[this.rep ID, any Data]],false)
 |res = (this.rep Iter[unknown Node[this.rep ID, any Data]],false)
```

Figure 4.7: Visualization of the typing process for the expression `map.iter` in the `Map` example from Figure 4.6.

safety, the `unknown` modifier allows us to keep the more precise `this.rep`. But since the boolean part of the result is *false* (due to the `unknown` modifier for the type argument), it is impossible to use the expression `map.iter` as left-hand side for an assignment.

```
select(this.rep Map[this.rep ID, any Data], node, this.rep Node[K, V]) = res
 |  calculating (Ttmp, btmp) ...
 |    subst(this.rep Node[K, V],
 |     |    {(this.rep Map[this.rep ID, any Data],this)}) = res
 |     |  calculating (u1, b1) ...
 |     |    subst1(this.rep, {(this.rep,this)}) = res
 |     |     |    assignable(this.rep, this.rep) = false
 |     |     |res = (this.rep.rep,false)
 |     |  (u1, b1) = (this.rep.rep,false)
 |     |  calculating T2 ...
 |     |    subst(K, {(this.rep Map[this.rep ID, any Data],this)}) =
 |     |       (this.rep ID,true)
 |     |    subst(V, {(this.rep Map[this.rep ID, any Data],this)}) =
 |     |       (any Data,true)
 |     |  T2 = {(this.rep ID,true), (any Data,true)}
 |     |  Tres = {(this.rep ID,true), (any Data,true)}
 |     |res = (this.rep.rep Node[this.rep ID, any Data],false)
 |  (Ttmp, btmp) = (this.rep.rep Node[this.rep ID, any Data],false)
 |res = (this.rep.rep Node[this.rep ID, any Data],false)

select(this.rep Iter[unknown Node[this.rep ID, any Data]], current, X) = res
 |  calculating (Ttmp, btmp) ...
 |    subst(X, {(this.rep Iter[unknown Node[this.rep ID, any Data]],this)}) =
 |       (unknown Node[this.rep ID, any Data],false)
 |  (Ttmp, btmp) = (unknown Node[this.rep ID, any Data],false)
 |res = (unknown Node[this.rep ID, any Data],false)
```

Figure 4.8: Visualization of the typing process for the expressions `map.node` and `i.current` in the `Map` example from Figure 4.6.

The typing for the expression `map.node` is very similar (Figure 4.8, the first block), while for the expression `i.current` the `unknown` modifier is propagated (Figure 4.8, the second block). Both expressions can be referenced by a variable of type `any Node[this.rep ID, any Data]`, as indicated on lines 22 and 23 in Figure 4.6. Assignments to `map.node` are disallowed since the main modifier for the resulting expression, `this.rep.rep`, is not precise enough. And assignments to `i.current` are disallowed since the main modifier is `unknown`.

Consider now the field selections on lines 27 and 28 in Figure 4.6. The resulting types for these selections (Figure 4.9) are computed by simply substituting the corresponding type argument for the type variables. Since this step is lossless, assignments to these fields could be allowed. They are, however, forbidden by the owner-as-modifier discipline, since the main modifier for the receiver `n`, which is `any`, does not allow field modifications. With regards to the ownership topology, assignments would be safe.

Finally for completeness, Figure 4.10 defines the `map` reference as an immutable value, as opposed to the mutable variable used in Figure 4.6. The resulting type for `map.iter` is then the more precise `this.rep Iter[map.rep Node[this.rep ID, any Data]]` as the typing process in Figure 4.11 shows. This allows to update the `iter` field of `map`, as line 6 indicates.

```
select(this.rep.rep Node[this.rep ID, any Data], key, K) = res
 |  calculating (Ttmp, btmp) ...
 |    subst(K, {(this.rep.rep Node[this.rep ID, any Data],this)}) =
 |       (this.rep ID,true)
 |  (Ttmp, btmp) = (this.rep ID,true)
 |res = (this.rep ID,true)

select(this.rep.rep Node[this.rep ID, any Data], value, V) = res
 |  calculating (Ttmp, btmp) ...
 |    subst(V, {(this.rep.rep Node[this.rep ID, any Data],this)}) =
 |       (any Data,true)
 |  (Ttmp, btmp) = (any Data,true)
 |res = (any Data,true)
```

Figure 4.9: Visualization of the typing process for the expressions `n.key` and `n.value` in the `Map` example from Figure 4.6.

```
object client extends Application {
    var id = new this.rep ID
    var data: any Data = new this.rep Data
    val map = new this.rep Map[this.rep ID, any Data] (id, data)

    var i = map.iter
    map.iter = i    // ok
}
```

Figure 4.10: `Map` example with the `map` reference being immutable.

```
select(map Map[this.rep ID, any Data], iter,
 |     this.peer Iter[this.rep Node[K, V]]) = res
 |  calculating (Ttmp, btmp) ...
 |    subst(this.peer Iter[this.rep Node[K, V]],
 |      |    {(map Map[this.rep ID, any Data],this)}) = res
 |      |  calculating (u1, b1) ...
 |      |    subst1(this.peer, {(map,this)}) = res
 |      |     |res = (this.rep,true)
 |      |  (u1, b1) = (this.rep,true)
 |      |  calculating T2 ...
 |      |    subst(this.rep Node[K, V],
 |      |      |    {(map Map[this.rep ID, any Data],this)}) = res
 |      |      |  calculating (u1, b1) ...
 |      |      |    subst1(this.rep, {(map,this)}) = res
 |      |      |     |res = (map.rep,true)
 |      |      |  (u1, b1) = (map.rep,true)
 |      |      |  calculating T2 ...
 |      |      |    subst(K, {(map Map[this.rep ID, any Data],this)}) =
 |      |      |       (this.rep ID,true)
 |      |      |    subst(V, {(map Map[this.rep ID, any Data],this)}) =
 |      |      |       (any Data,true)
 |      |      |  T2 = {(this.rep ID,true), (any Data,true)}
 |      |      |  Tres = {(this.rep ID,true), (any Data,true)}
 |      |      |res = (map.rep Node[this.rep ID, any Data],true)
 |      |  T2 = {(map.rep Node[this.rep ID, any Data],true)}
 |      |  Tres = {(map.rep Node[this.rep ID, any Data],true)}
 |      |res = (this.rep Iter[map.rep Node[this.rep ID, any Data]],true)
 |  (Ttmp, btmp) = (this.rep Iter[map.rep Node[this.rep ID, any Data]],true)
 |res = (this.rep Iter[map.rep Node[this.rep ID, any Data]],true)
```

Figure 4.11: Visualization of the typing process for the expression `map.iter` with immutable `map` reference as in the example in Figure 4.10.

4.3 First-class functions

Scala supports the concept of *first-class functions*, i.e. functions that may be passed as parameters or returned as results. Functions taking other functions as parameters or returning them as results are called *higher-order functions*.

The type of a function in Scala is denoted by `(T1, ..., Tn) => U` where the $\texttt{T}_i$ represent the argument types and `U` the result type. These function types are shorthands for the class types `Function`n`[T1, ..., Tn, U]` that define `apply` methods. The presence of such an `apply` method in a class allows to apply an instance of this class to a matching parameter list in Scala. Figure 4.12 presents a small example.

```
class fun extends Function1[Object, String] {
  def apply (o: Object): String = o.toString
}

//...
val f: (Object => String) = new fun
f(new Object)
```

Figure 4.12: Applying objects.

Higher-order functions in Scala are therefore functions taking parameters of a `Function`n type or returning results of such a type. Anonymous functions are passed by wrapping their body in an anonymous `Function`n instance and class methods can be mapped to `Function`n instances as well through Eta-expansion (Scala Language Specification [18], Section 6.24.5).

The Universe types for functions is obtained by annotating the `Function`n types and can be directly applied to the shorthands as well. For example the type `this.peer Function1[this.rep Object, any String]` translates to `this.peer (this.rep Object => any String)`.

The default value for the main modifier is `this.peer` for anonymous functions and `p.peer` for a function mapping a member method of an object instance reachable through path p. This allows function passing to retain as much of its expressiveness as possible. Since a function that has been created by mapping a member method, can have side-effects on the instance of the class containing the method, it is not safe to use `any` as main modifier and assume purity for the `apply` methods of functions. Similarly for anonymous functions which can have effects on non-local values as well.

There is, however, still room for improvements regarding the handling of anonymous functions. Examples include explicit purity annotation for anonymous functions, to be able to map them to subtypes of the existing `Function`n types with pure `apply` methods. Also pure member methods should be mapped to such pure functions as well.

4.4 Summary

The static precision of the Universe type system is clearly profiting from path-dependent types. For instance they allow us to read `rep` fields on receivers other than `this` without having to resort to `any` to type the resulting expression. If the receiver for such a field selection is immutable then the resulting expression will have a type with an exact path indicating where this object is obtained from. This for example helps identifying the collection an iterator is working on and allows to avoid casts when passing the object back to its owner for modifications.

A great deal of flexibility is also owed to the loose paths which for example allow to use the same iterator variable for all collections owned by peers of `this` and only those.

Path-dependent Universe Types also allow to define more flexible encapsulation policies. For example it is possible to allow an object to create instances in the context of another object. The abstract factory pattern is an example where [16] identified the need for such object creations in foreign contexts. Path-dependent types would allow a factory to provide a method, which, when

called from an object `o` with a reference to `o` itself as argument would then create the products directly in `o`'s context and return a reference to it. Additional rules would have to ensure that only the owner of a context can allow other objects to create objects in its context. For example by annotating the parameter of the `create` method of the factory and only allowing `this` as value for the corresponding argument.

Regarding parameterized types we believe that the variance annotations for type parameters as used by Scala present a much more elegant way of dealing with subtyping for parameterized types than the wildcard concept used by Java. Practical usage of co- and contravariant parameterized Universe types will, however, have to be evaluated with case studies or real world applications.

Chapter 5

Related Work

Erik Ernst builds on dependent types to take polymorphism to the multi-object level. Family Polymorphism [12] allows to declare and manage relations between several classes polymorphically. A number of classes is grouped to a family by means of a family object. The relation of the dependent classes to the identity of this family object guarantees type safety in the sense that classes of different families are not mixed. This is in some sense very similar to the idea behind ownership type systems. The main difference is that only inner classes of the class of the family object qualify for families, while ownership systems generally do not apply such restrictions.

With Higher-order Hierarchies [13] he then takes the concept of class families to the next level by investigating inheritance for entire object structures.

Similarly Igarashi and Viroli [15] use relative path types to preserve relationships between members inside a class family through extension. They adopt they idea of exact types to improve static type safety and add support for controlling the mixing of class families in a safe way. Class families in their FJ_{path} calculus are therefore not strictly coupled to nesting anymore.

The idea of exact and inexact types is in many ways similar to our stable, exact and loose paths. Depending on the receiver type unsafe operations can be statically forbidden. In their case, for example, a method taking a relative parameter can only be called if the receiver is exact. Along the same lines we can only allow method calls if the parameter context, as seen by the receiver, can be matched with an actual argument object relative to the viewpoint of the caller for all parameters.

Simple Loose Ownership Domains [23] simplify Ownership Domains [1] by reducing the number of domains and hardwiring the access permissions between domains. They are then introducing loose domains which are very similar to our loose paths. For instance they allow to overcome the Ownership Domains restriction of accesses to foreign public domains having to go through a final reference to the object owning the accessed domain.

Tribe [5] supports path types depending on both classes and objects. It can therefore distinguish objects from different instances of a family and is not limited to class based families. With its `out` reference it partly adopts the concept of topological type systems, which we are basing our work on. Compared to earlier approaches this reduces the need to pass family objects and for example lets the system support method arguments depending on each other, which we also support. As one might expect the designers of Tribe were facing some problems with regards to type compatibility of types which are not syntactically equivalent, just like we did.

Another work by Clarke and Drossopoulou [4] uses effects shapes to describe collections of contexts. Our loose paths allow us to do similar things. Their *band* shape can be obtained by concatenating a number of `rep` modifiers and the *under* shape corresponds to the `reps` modifier, possibly prefixed with a sequence of `rep` modifiers. The shapes they used support numbering to indicate the level or starting level of a shape. We do not have a similar construct in our language, since we believe long modifier sequences will be rarely used in real world programs. However, it is just syntactic sugar and can easily be added if there is a demand. Then again our `up` and `ups` modifiers also allow us to have *band* shapes with negative indices and something like an *above*

shape that includes the contexts of an objects transitive owners.

Besides shapes they also provide support for creation time ownership transfer similar to the one we are presenting. Yet another way to approach this would be using uniqueness. Takano [27] investigated this problem and presented an application of uniqueness and ownership transfer to the Universe Type System.

Cunningham et al. [7] use Universes and paths to statically identify race conditions in multi-threaded programs. We believe that having inherent support for paths in the type system can only improve such approaches.

Chapter 6

Conclusion

In this chapter we summarize the contributions of our system and give an outlook to possible directions for future work.

6.1 Contribution

We have presented a first formalization for a type system combining ownership and alias control in the form of the owner-as-modifier discipline with path-dependent types and type genericity. The increased precision introduced by the path-dependent Universe Types make room for investigating even more flexible approaches for encapsulation policies to control aliasing.

6.2 Implementing path-dependent Universe Types

It would be interesting to have an actual implementation of our type system. Possibilities include, amongst other, integration into the main Scala compiler. It is, however, not limited to Scala or Scala based projects. While we have based our type system on Scala, path-dependent Universe types are in fact not depending on any Scala specific feature. Therefore the main part of the type system could easily be applied to other languages such as Java or C#. Only the extension to parameterized types presented in Section 3.2 is linked to specific Scala features, namely the support for variance annotations. It should thus be possible to implement our type system as an extension of the existing Universe compiler in the MultiJava project[1] or use a pluggable type system for Java like the one described by Andreae et al. [2]. The later, however, would require annotations not to be limited to declarations. Instead we would need to be able to annotate types. There are actually efforts in the Java community process to adjust the scope of annotations accordingly.

Scala already supports type annotations to some extent and recent work on pluggable type systems for the Scala compiler makes an implementation of the Universe type system for Scala very attractive. There are chances that the prototype implementation can be mapped to a compiler plugin with a reasonable effort. Pluggable Universe Types for Scala would also ease the usage of the system, since the support for pluggable types is most likely being integrated into the standard Scala compiler.

6.3 Future Work

Besides completing our system to a complete formal specification with type safety proof, we are thinking about further extending the expressiveness of our loose paths by adding some form of class dependent types. The Tribe [5] calculus for example already supports this.

[1] `http://multijava.sourceforge.net/`

Inspired by Ownership Domains [1] we are also thinking about introducing support for arbitrary universes. This would allow objects to define several distinct representation contexts as opposed to the single `rep` context that is supported now. It would however require to have the context modifiers qualified by the class they are defined in. This would then extend the expressiveness of reference types to not only define context and class of an object but also the contexts and classes of objects on the path. In other words it would bundle related objects to some kind of families and all in all lead in a direction similar to what can be done with variant path types [15] for example.

```
class Object {
  rep: Universe
}

class C {
  data: Universe
  f: this.data E
}

class E {
  store: Universe
  x: this.store C
  g: this.store#C.data E
}
```

Figure 6.1: Pseudocode to illustrate the idea of arbitrary universes.

In pseudocode this could look somewhat like the example in Figure 6.1. As a follow-up it would maybe be interesting to be able to define different rights on the user-defined universes to allow a more fine grained control of actions on and with objects. Again Ownership Domains [1] supports this already to some extent. Yet introducing access control based on universes leaves much room for experiments with flexible encapsulation policies for ownership type systems. Examples include permitting objects to have write access on their entire transitive representation `this.reps` instead of only the first level `this.rep`. Or by allowing write access to `this.ups` an "emulation" of Ownership Types [3] could be established. Also one could restrict the passing of `rep` objects by never allowing them to be assigned or passed as `any` references and many more. Control of such features can be achieved in different ways. One being compiler switches, another one annotations or even by using a pluggable type system [2].

Finally the relation of the `unknown` and `any` modifiers are not satisfying yet, which we want to improve and we also need to further investigate the possibilities for the choice of a context for singleton objects.

Appendix A

The `assignable` function

This appendix is concerned with the *assignable* function (Figure 3.7) used for the viewpoint adaptation performed by *subst* (Figure 3.6) to type field selections (Section 3.1.4) and method calls (Section 3.1.5). Since method calls can be seen as a special case of field selections, we are explaining it for field selections here and leave the application to method calls to the reader.

```
class C {
  val f: ?? Object
  var g: ?? Object
}

val a: ?? C
var b: ?? C
```

Figure A.1: Code snippet to discuss field selections.

Consider Figure A.1 for the declaration of a class `C` with two fields of unspecified universe and two references to instantiations of the class. We will now discuss all four combinations of field selections possible for this example code and explain what universes are admissible in order to get a combined universe which does not loose any information. For this the universes marked with question marks in the source code are ranging over $UPath$. The `unknown` and `any` universes are not of interest since their handling is trivial. In order to simplify the explanations we restrict the field paths to all valid paths obtained by combining the start value `this` with a sequence of one or more context modifiers. We are discussing the extension to more elaborate field paths after the main discussion. The receiver path can, without loss of generality, be seen as a stable prefix p followed by a sequence of one or more context modifiers.

Accessing fields on an immutable receiver The case of both, receiver and field being immutable is obviously not very interesting since it does not involve a viewpoint adaptation and therefore does not have to be handled by *assignable*. If only the field is mutable it is a very similar case. The resulting universe is then a simple combination of the stable receiver path and the field path and does not loose any information either.

Accessing fields on a mutable receiver If the receiver is mutable it does not matter whether the field is mutable or not. Of course immutable fields cannot be assigned, even if *assignable* returns *true*. But the resulting universe is computed exactly the same way.

The general idea is as follows: we cannot be certain about the identity of the receiver object, but the field universe is relative to this receiver object. So in order to match the context identified by the combined universe of field and receiver, we need to eliminate that uncertainty. In other

words: if the field universe does not depend on the exact identity of the receiver but only on the context the receiver is part of, the viewpoint adaptation can be lossless and we might be able to reach this context with a path starting from p.

receiver path (b)	field path (g)	combined path (b.g)
p.peer	this.peer.$\overline{\texttt{rep}}$	p.peer.$\overline{\texttt{rep}}$
	this.up.$\overline{\texttt{up}}$.$\overline{\texttt{rep}}$	p.up.$\overline{\texttt{up}}$.$\overline{\texttt{rep}}$
	this.ups.$\overline{\texttt{rep}}$	p.ups.$\overline{\texttt{rep}}$
p.rep	this.peer.$\overline{\texttt{rep}}$	p.rep.$\overline{\texttt{rep}}$
	this.up.$\overline{\texttt{up}}$.$\overline{\texttt{rep}}$	p.peer.$\overline{\texttt{rep}}$ or p.$\overline{\texttt{up}}$.$\overline{\texttt{rep}}$
	this.ups.$\overline{\texttt{rep}}$	p.rep.ups.$\overline{\texttt{rep}}$
p.up	this.peer.$\overline{\texttt{rep}}$	p.up.$\overline{\texttt{rep}}$
	this.up.$\overline{\texttt{up}}$.$\overline{\texttt{rep}}$	p.up.up.$\overline{\texttt{up}}$.$\overline{\texttt{rep}}$
	this.ups.$\overline{\texttt{rep}}$	p.up.ups.$\overline{\texttt{rep}}$
p.reps	-	-
p.ups	-	-

Figure A.2: Admissible field paths for mutable receivers: simple cases.

Figure A.2 lists the admissible field paths for the five most basic receiver paths and also shows the respective combined path. If the receiver path contains a reps or ups modifier, it is impossible to know the exact context of the receiver and there is no admissible field path. All of the other three cases can be covered with the same three field paths. Namely if the field path starts with a peer modifier, the field just relates to the the context of the receiver, not the receiver itself. If it starts with an up or ups modifier, it depends on the context of the receiver's owner. It cannot start with rep or reps since these modifiers relate to the receiver object. If it starts with up, it may be followed by a sequence of up modifiers where the last element may be an ups modifier. And all three field paths may end with a sequence of rep modifiers, where again the last element may be a reps modifier. Note that according to the path simplification rules further peer modifiers would be redundant as would be an up or ups modifier following the peer modifier in the first field path.

receiver path (b)	field path (g)	combined path (b.g)
p.up.$\overline{\texttt{up}}_x$	this.peer.$\overline{\texttt{rep}}$	p.up.$\overline{\texttt{up}}_x$.$\overline{\texttt{rep}}$
	this.up.$\overline{\texttt{up}}_z$.$\overline{\texttt{rep}}$	p.up.up.$\overline{\texttt{up}}_{x+z}$.$\overline{\texttt{rep}}$
	this.ups.$\overline{\texttt{rep}}$	p.up.$\overline{\texttt{up}}_x$.ups.$\overline{\texttt{rep}}$
p.peer.rep.$\overline{\texttt{rep}}_x$	*this*.up.$\overline{\texttt{up}}_y$.$\overline{\texttt{rep}}$	p.peer.$\overline{\texttt{rep}}$ or p.up.$\overline{\texttt{up}}_{y-1}$.$\overline{\texttt{rep}}$
p.rep.rep.$\overline{\texttt{rep}}_x$	this.up.$\overline{\texttt{up}}_y$.$\overline{\texttt{rep}}$	p.rep.$\overline{\texttt{rep}}$ or p.peer.$\overline{\texttt{rep}}$ or p.up.$\overline{\texttt{up}}_{y-2}$.$\overline{\texttt{rep}}$
p.up.$\overline{\texttt{up}}_z$.rep.$\overline{\texttt{rep}}_x$	this.up.$\overline{\texttt{up}}_y$.$\overline{\texttt{rep}}$	p.up.$\overline{\texttt{up}}_{z+y-x}$.$\overline{\texttt{rep}}$

Figure A.3: Admissible field paths for mutable receivers: complex cases. x, y, and z indicate the cardinality of the modifier sequence and $y \geq x$.

Figure A.3 is listing the more complex cases. For the case of the receiver path being a sequence of up modifiers, the argumentation goes along the same lines as for the simple cases above. And if the receiver path ends with a sequence of length x of rep modifiers, the field path has to start with at least x up modifiers in order to identify a context that can also be reached from p without loss of precision.

Special cases There are two special cases for mutable receivers which are not covered so far. First if the field path is depending on a singleton object rather than the receiver object, the transformation yields the field path again and thus does not loose any information. This case is already covered by *subst*1 and therefore does not have to be handled here.

Second if for example the path of field `g` depends on another field of the same object, for example `f`, the viewpoint adaptation will be lossy. For mutable receivers we can not preserve such dependencies, unless the path of the last field in the dependency chain does not depend on the current object `this` but on some singleton object. Keep in mind that path normalization eliminates trivial dependencies like for example field `g` being `peer` to field `f`. Since this case is already covered by *subst*1 as well, the limitation of the field paths having to consist of `this` and a sequence of context modifiers for the cases covered by *assignable* is reasonable.

Appendix B

Predicate and Function Overview

In the following we list the signatures of the predicates and functions used by our type system and briefly explain the meaning of the parameters. The intention of this chapter is to support the understanding of some function definition or type rule by reducing the need to look up definitions of other functions used in this type rule or function.

$\Gamma :: Start \rightarrow Type$
$\Gamma(s) = T$
s : start value/variable
T : type of s

$normalize :: SUniv \rightarrow Univ$
$normalize(suniv) = u$
$suniv$: user specified (degenerated) universe annotation
u : normalized universe

B.1 Functions concerned with universes

$isStable :: Univ \rightarrow bool$
$isStable(u) = b$
u : universe
b : $true$ if u is stable, $false$ otherwise

$isExact :: Univ \rightarrow bool$
$isExact(u) = b$
u : universe
b : $true$ if u is exact, $false$ otherwise

$subst1 :: Univ \times \overline{Univ \times Val} \rightarrow Univ \times bool$
$subst1(u_0, \overline{u\ x}) = (u_{res}, b_{res})$
u_0 : universe of the expression that needs to be adapted to a different viewpoint
$\overline{u\ x}$: sequence of pairs defining the viewpoints
where u_i is the declared universe of value x_i
u_{res} : resulting universe of the substitution
b_{res} : $true$ if type information got preserved and assignments are safe

$assignable :: Univ \times Univ \rightarrow bool$
$assignable(u_0, u_1) = b$
u_0 : universe of the reciever expression
u_1 : universe of the field
b : $true$ if assignments to a field annotated with u_1
on a reciever with universe u_0 can be permitted

$loosen :: UPath \rightarrow Univ$
$loosen(p) \quad = \quad u_{res}$
p : path
u_{res} : p with last component of stable prefix loosened

$simplify :: Univ \rightarrow Univ$
$simplify(u) \quad = \quad u_{res}$
u : universe
u_{res} : the most simple universe describing the exact same universe or set of universes as u

$simplify1 :: UPath \rightarrow UPath$
$simplify1(p) \quad = \quad p_{res}$
p : path
p_{res} : p with one redundant context modifier less

$simplify2 :: UPath \rightarrow UPath$
$simplify2(p) \quad = \quad p_{res}$
p : path
p_{res} : p with the last (redundant) field selection removed

$isModifieable :: Univ \rightarrow bool$
$isModifieable(u) \quad = \quad b$
u : universe
b : *true* if u is modifieable when adhering to the owner-as-modifier discipline

B.2 Functions concerned with types

$setMain :: NType \times Univ \rightarrow NType$
$setMain(N, u) \quad = \quad N_{res}$
N : non-variable type
u : universe
N_{res} : N with the main modifier set to u

$select :: NType \times Field \times Type \rightarrow Type \times bool$
$select(T_0, f, T_1) \quad = \quad (T_{res}, b_{res})$
T_0 : declared type of the reciever expression
f : field identifier
T_1 : declared type of the field
T_{res} : combined type of the field selection
b_{res} : *true* if type information got preserved and assignments are safe

$subst :: Type \times \overline{Type \times Val} \rightarrow Type \times bool$
$subst(T, \overline{T_x\ x}) \quad = \quad (T_{res}, b_{res})$
T : full type of the expression that needs to be adapted to a different viewpoint
$\overline{T_x\ x}$: sequence of pairs defining the viewpoints where T_{x_i} is the declared type of value x_i
T_{res} : resulting type of the substitution
b_{res} : *true* if type information got preserved and assignments are safe

B.3 Subtyping

$<:_u :: Univ \times Univ \rightarrow bool$
$u_0 <:_u b \quad = \quad b$
u_0, u_1 : universes
b : *true* if u_0 is a subtype of u_1, *false* otherwise

$<: :: Type \times Type \rightarrow bool$
$T_0 <: T_1 \quad = \quad b$
$T_0, T_1 \quad : \quad$ types
$b \quad : \quad true$ if T_0 is a subtype of T_1, $false$ otherwise

$<:_{arg} :: Type \times Type \rightarrow bool$
$T_0 <:_{arg} T_1 \quad = \quad b$
$T_0, T_1 \quad : \quad$ types
$b \quad : \quad true$ if T_0 is a subtype of T_1 according to the limited covariant subtyping for typearguments,$false$ otherwise

Appendix C

Prototype implementation

C.1 universes.scala

```
0  package universes.formalizator;

   import Coercions.Iterator2String
   import scala.collection.immutable.EmptyMap

5  /**
    * Base class for universes.
    *
    * @author Daniel Schregenberger
    */
10 abstract class Univ {
     /**
      * Construct a new path by appending f to the list of field selections
      * for this path.
      *
15    * @param _f the field to append.
      *
      * @return the new path
      *
      * @throws error if this is not a path (but unknown or any) or the path
20    *         already contains a context sequence.
      */
     def ° (_f: Field): UPath = error("cannot append field to '" + this + "'")

     /**
25    * Construct a new path by appending c to the list of contexts for this
      * path.
      *
      * @param _c the context to append.
      *
30    * @return the new path
      *
      * @throws error if this is not a path (but unknown or any).
      */
     def ° (_c: Context): UPath =
35                       error("cannot append context to '" + this + "'")

     /**
      * Construct a new path by appending f to the list of field selections
```

```
     * for this path.
     * Alias for °.
     *
     * @param _f the field to append.
     *
     * @return the new path
     *
     * @throws error if this is not a path (but unknown or any) or the path
     *         already contains a context sequence.
     */
    def + (_f: Field) = this°_f

    /**
     * Construct a new path by appending c to the list of contexts for this
     * path.
     * Alias for °.
     *
     * @param _c the context to append.
     *
     * @return the new path
     *
     * @throws error if this is not a path (but unknown or any).
     */
    def + (_c: Context) = this°_c

    /**
     * Construct a Universe type by annotating i with this universe.
     *
     * @param i the type to annotate.
     *
     * @return the Universe type with normalize(this) as universe.
     */
    def ## (i: Instance) = new NType(this.normalize, i.clazz, i.typeParams)

    /**
     * Construct a Universe type by annotating i with this universe.
     * Does not normalize the universe but only simplifies it.
     * Used to generate degenerate types for the testcases
     * (for expected results).
     *
     * @param i the type to annotate.
     *
     * @return the Universe type with normalize(this) as universe.
     */
    def ### (i: Instance) = new NType(this.simplify, i.clazz, i.typeParams)

    /**
     * Construct a Universe type by annotating i with this universe.
     * Alias for ##.
     *
     * @param i the type to annotate.
     *
     * @return the Universe type with normalize(this) as universe.
     */
    def ++ (i: Instance) = this ## i

    /**
     * Construct a Universe type by annotating i with this universe.
```

```
     * Does not normalize the universe but only simplifies it.
     * Used to generate degenerate types for the testcases
     * (for expected results).
100  * Alias for ###.
     *
     * @param i the type to annotate.
     *
     * @return the Universe type with normalize(this) as universe.
105  */
    def +++ (i: Instance) = this ### i

    /**
     * Indicates if the path is stable.
110  *
     * @return true if the path is stable, false otherwise.
     */
    def isStable = false

115 /**
     * Indicates if the path is exact.
     *
     * @return true if the path is exact, false otherwise.
     */
120 def isExact = false

    /**
     * Normalizes paths and rejects invalid universes.
     *
125  * @return the normalized path.
     *
     * @throws error if path is invalid.
     */
    def normalize : Univ = normalize(true)
130

    /**
     * Normalizes paths and rejects invalid universes.
     * Issues a warning when the normalization was lossy.
     * For internal use.
135  *
     * @param toplevel indicates if the current recursion is at
     * the toplevel (helps reduce amount of warnings).
     *
     * @return the normalized path.
140  *
     * @throws error if path is invalid.
     */
    def normalize (toplevel: boolean) = this

145 /**
     * Performs viewpoint adaptation of a universe with regards
     * to another universe. The start value of this path will be
     * replaced by the universe in the matching pair of the
     * viewpoint list.
150  *
     * @param viewpoints List of universe-value pair representing
      *         the viewpoint(s).
     *
     * @return the viewpoint adapted universe.
```

```
155     *
        * @throws error if more than one value matches.
        */
       def subst1 (viewpoints: List[(Univ, Val)]) = (this, true)

160    /**
        * Checks if the context (or set of contexts) described by u
        * as seen from viewpoint 'this' can be described with the
        * same exactness from the viewpoint of this path.
        * Used for example to check assignments to fields on mutable
165     * recievers.
        *
        * Assumes paths are valid / normalized.
        *
        * @param u the universe of the field
170     *
        * @return true if the combined context does not loose any
         *         information, false otherwise.
        */
       def assignable (u: Univ) = true
175
       /**
        * Simplifies paths by removing redundant context modifiers,
        * ordering and normalizing the context modifier sequence
        * and removing path aliasing by trying to apply loosen
180     * and simplify the result. If the result of loosen cannot
        * be simplified it is undone, otherwise an equivalent but
        * simpler path is found.
        *
        * @return the most simple universe or path equivalent to this.
185     */
       def simplify = this

       /**
        * The subtyping operator for universes.
190     *
        * @param u the universe to test for being supertype of this.
        *
        * @return true if this <: u, false otherwise.
        */
195    def <:< (u: Univ) = (u == Any) || (u == Unknown) || (this == u)
   }


   /**
200 * The 'unknown' universe.
    *
    * @author Daniel Schregenberger
    */
   object Unknown extends Univ {
205    /** String representation (== 'unknown'). */
       override def toString() = "unknown"
   }


210 /**
    * The 'any' universe.
    *
```

```
 * @author Daniel Schregenberger
 */
object Any extends Univ {
  /** String representation (== 'any'). */
  override def toString() = "any"

  override def <:<(u: Univ) = (u == Any)
}


/**
 * Class for universe paths.
 *
 * @param s the start value/variable.
 * @param f list of field selections with f.head being the first
 *        ( the selection on s).
 * @param c list of contexts with same ordering.
 *
 * @author Daniel Schregenberger
 */
final case class UPath(s: Start, f: List[Field], c: List[Context]) extends Univ {
  /**
   * Alternative constructor.
   *
   * @param _s the start value/variable.
   * @param _f a field selection.
   * @param _c a context.
   */
  def this (_s: Start, _f: Field, _c: Context) =
    this(_s, _f :: Nil, _c :: Nil)

  /**
   * Convert to string in the form: s.f.c
   *
   * @result the String representation of the path.
   */
  override def toString = (c foldLeft ("" +
      ( f foldLeft s.toString) {(x, y) => x + "." + y})
    ) {( x, y) => x + "." + y}


  /* see Univ for documentation or use the generated ScalaDoc */
  override def ° (_f: Field) = this.c match {
    case Nil => new UPath(s, f ::: (_f :: Nil), Nil).simplify
    case _ => error("cannot append field to path with contexts")
  }
  /* see Univ for documentation or use the generated ScalaDoc */
  override def ° (_c: Context) = new UPath(s, f, c ::: (_c :: Nil)).simplify

  /* see Univ for documentation or use the generated ScalaDoc */
  override def isStable = s match {
    case Val(_,_) => (c == Nil) && !f.exists(_.isMutable)
    case _ => false
  }

  /* see Univ for documentation or use the generated ScalaDoc */
  override def isExact = (c.length == 1) && new UPath(s, f, Nil).isStable

```

```
    /* see Univ for documentation or use the generated ScalaDoc */
    override def normalize (toplevel: boolean) = {
        if ( s == null ) {
            if ( c == Nil ) {
                if ( f == Nil ) {
                    new UPath(This, Nil, Peer :: Nil)
              }   else
                    error("Invalid universe -- " +
                     "   at least one of s, f or c must be specified.")
          }  else
                new UPath(This, f, c).normalize(toplevel)
      }  else {
            if ( c == Nil ) {
                error("Invalid universe: " + this +
                 " --   user specified paths must end with a context modifier.")
          }  else {
                if ( new UPath(s, f, Nil).isStable )
                    this.simplify
                else {
                    val res = this.loosen.simplify.normalize(false)
                    if ( toplevel )
                        logger.warn("normalizing '" + this + "' to '" + res + "'")
                    res
                }
            }
      }
    }

    /* see Univ for documentation or use the generated ScalaDoc */
    override def subst1 (viewpoints: List[(Univ, Val)]) = {
      ( viewpoints filter {_._2 == s}) match {
            case Nil => (this.simplify, true)
            case (ui, xi ) :: Nil => {
                logger("subst1(" + this + ", " +
                       Iterator2String(viewpoints.elements) + ") = res").++(" | ")
                val res = (ui match {
                    case Unknown => (Unknown, false)
                    case Any => (Unknown, false)
                    case path: UPath => {
                        if ( path.isStable )
                        (    new UPath(path.s, path.f ::: f, c).simplify, true)
                        else if ( ( this.f == Nil) )
                        (    new UPath(path.s, path.f, path.c ::: c).simplify,
                             path.assignable(new UPath(This, Nil, c).simplify))
                        else
                        (    this.loosen.simplify.subst1(viewpoints)._1, false)
                    }
                })
                logger.--("res = " + res).--
                res
            }
            case _ => error("more than one matching value for viewpoint adaptation")
      }
    }

    /* see Univ for documentation or use the generated ScalaDoc */
    override def assignable (fieldUniv: Univ) = {
        /**
```

```
        * Checks if a context modifier sequence contains only rep
330      * modifiers and at most one reps modifier at the end.
        *
        * @param seq the context modifier seqence to check.
        *
        * @return true if seq contains only rep modifiers and at
335       *         most one reps modifier at the end. false otherwise.
         */
        def isRepSequence (seq: List[Context]) = seq.reverse match {
            case Nil => true
            case Rep :: tail  => tail.forall(_ == Rep)
340         case Reps :: tail  => tail.forall(_ == Rep)
            case _ => false
        }

        // assignable(p.ups(.c)*, _) and assignable(p.reps(.c)*, _)
345     val res = !c.contains(Ups) && !c.contains(Reps) &&
            new UPath(s, f, Nil).isStable && (fieldUniv match {
                /* assignable(p.c(.up)*, this.peer) */
                case UPath(This, Nil, Peer :: Nil) =>
                    ((c.length == 1) || c.forall(_ == Up))
350             /* assignable(p.c(.up)*, this.peer(.rep)*) */
                case UPath(This, Nil, Peer :: tail) =>
                    ((c.length == 1) ||
                     c.forall(_ == Up)) && isRepSequence(tail)
                case UPath(This, Nil, fieldContexts) =>
355                     (c.reverse, fieldContexts) match {
                    /* assignable(p.peer, this.up(.c)*) */
                    case (Peer :: Nil, Up :: _) => true
                    /* assignable(p.peer, this.ups(.c)*) */
                    case (Peer :: Nil, Ups :: _) => true
360                 /* assignable(p.rep, this.up(.c)*) */
                    case (Rep :: Nil, Up :: _) => true
                    /* assignable(p(.up)*.rep, this(.up)+(.rep)*) */
                    case (Rep :: t, Up :: tt) =>
                        (t.length == 1) ||
365                     (t.forall(_ == Up) && isRepSequence(tt)) ||
                        new UPath(s, f, t.reverse).assignable(
                            new UPath(This, Nil, tt))
                    /* assignable(p.rep, this.ups(.c)*) */
                    case (Rep :: Nil, Ups :: _) => true
370                 /* assignable(p(.up)*.rep, this.ups(.rep)*) */
                    case (Rep :: t, Ups :: tt) =>
                        (t.length == 1) ||
                        (t.forall(_ == Up) && isRepSequence(tt)) ||
                        new UPath(s, f, t.reverse).assignable(
375                             new UPath(This, Nil, tt))
                    /* assignable(p(.up)+, this.up(.c)*) */
                    case (Up :: t, Up :: _) => t.forall(_ == Up)
                    /* assignable(p(.up)+, this.ups(.c)*) */
                    case (Up :: t, Ups :: _) => t.forall(_ == Up)
380                 case _ => false
                }
            })
        logger.++.apply("assignable(" + this + ", " + fieldUniv + ") = " + res).--
        res
385 }
```

```
    /**
     * Replace the last field selection by the universe of the field.
     * The result is a supertype of the original path.
390  * If there are no more field selections, replace start value (!= this)
     * with its universe.
     *
     * @return the loosened path.
     */
395  def loosen = {
       if ( f.length > 0 ) {
         f.last.getType match {
           case n: NType => {
             n.univ.subst1((new UPath(s, f.init, Nil), This) :: Nil)._1 match {
400            case Unknown => Unknown
               case Any => if ( c.length > 0 ) Unknown else Any
               case UPath(fieldS, fieldF, fieldC) =>
                 new UPath(fieldS, fieldF, fieldC ::: c)
             }
405        }
           case _ => this
         }
       } else { s.getType match {
           case n: NType => n.univ match {
410          case Unknown => Unknown
             case Any => if ( c.length > 0 ) Unknown else Any
             case UPath(startS, Nil, startC) =>
               new UPath(startS, Nil, startC ::: c)
           }
415        case _ => this
         }
       }
     }

420  /* see Univ for documentation or use the generated ScalaDoc */
     override def simplify = {
       /**
        * Simplifies sequences of context modifiers by removing
        * redundant ones, ordering and normalizing them.
425     *
        * @param l sequence of context modifiers in reverse order
        *          (for more efficient recursions).
        *
        * @return the simplified context modifier sequence.
430     *          In reverse order as well.
        */
       def simplify1 (l: List[Context]) : List[Context] = l match {
         case Peer :: Peer :: p => Peer :: p
         case Up :: Peer :: p => Up :: p
435      case Ups :: Peer :: p => Ups :: p
         case Peer :: Rep :: p => Rep :: p
         case Up :: Rep :: p => Peer :: p
         case Peer :: Up :: p => Up :: p
         case Peer :: Ups :: p => Ups :: p
440      case Peer :: Reps :: p => Reps :: p

         case Up :: Ups :: p => Ups :: Up :: p
         case Rep :: Reps :: p => Reps :: Rep :: p
```

```
445             case Ups :: Ups :: p => Ups :: Up :: p
                case Reps :: Reps :: p => Reps :: Rep :: p

                case (c: Context) :: tail => c :: simplify1(tail)
                case x => x
450         }

            /*
             * Applies simplify1 until the result equals the inputand also
             * simplifies the cases involving the owner field and the first context.
455          *
             * @param path the path to simplify.
             *
             * @return the simplified path.
              */
460         def simplify11 (path: UPath) : UPath = {
                var old: List[Context] = Nil
                var cres = path.c.reverse
                while ( cres != old ) {
                    old = cres
465                 cres = simplify1(cres)
              }
                cres = cres.reverse

                if ( (path.f != Nil) && (cres != Nil) && (path.f.last == Owner) ) {
470                 cres.head match {
                        case Peer => simplify11(new UPath(path.s, path.f.init,
                                        (Up :: Nil ) ::: cres.tail))
                        case Rep => simplify11(new UPath(path.s, path.f.init,
                                        (Peer :: Nil ) ::: cres.tail))
475                     case _ => new UPath(path.s, path.f, cres)
                    }
              } else
                    new UPath(path.s, path.f, cres)
            }
480

            /**
             * Simplifies paths by removing path aliasing. It does so
             * by trying to apply loosen and simplify the result. If
             * the result of loosen cannot be simplified it is undone,
485          * otherwise an equivalent but simpler path is found.
             *
             * @param path the path to simplify.
             *
             * @return the most simple universe or path equivalent to
490          *         the argument.
              */
            def simplify2 (path: UPath) : UPath = {
                path.loosen match {
                    case p: UPath => {
495                     val res = simplify11(p)
                        if ( p != res )
                            simplify2(res)
                        else
                            path
500                 }
                    case _ => path
              }
```

```
        }

505     simplify2(simplify11(this))
    }

    /* see Univ for documentation or use the generated ScalaDoc */
    override def <:< (u: Univ) = u match {
510     case p: UPath => this <:< p
        case Any => true
        case Unknown => true
        case _ => false
    }
515

    /**
     * The subtyping operator for paths. Tries super.<:<(u: Univ) first.
     * Then calculates all direct supertypes of this. If one of these
     * equals path we are finished. Otherwise try x <:< path for each
520  * x in the list of direct supertypes.
     *
     * @param path the path to test for being supertype of this.
     *
     * @return true if this <: path, false otherwise.
525  */
    def <:< (path: UPath) : boolean = super.<:<(path) || {
        /**
         * Calculate next supertype for sequences of context modifiers
         * containing at least one rep modifier.
530      *
         * @param l sequence of context modifiers in reverse order
          *        (for more efficient recursions).
         *
         * @return the possibly modified context modifier sequence.
535       *         In reverse order as well.
          */
        def rep2reps (l: List[Context]) : List[Context] = l match {
            case Rep :: p => Reps :: p
            case Reps :: Rep :: p => Reps :: p
540         case (c: Context) :: tail => c :: rep2reps(tail)
            case x => x
        }

        /**
545      * Calculate next supertype for sequences of context modifiers
         * containing at least one up modifier.
         *
         * @param l sequence of context modifiers in reverse order
          *        (for more efficient recursions).
550      *
         * @return the possibly modified context modifier sequence.
          *         In reverse order as well.
          */
        def up2ups (l: List[Context]) : List[Context] = l match {
555         case Up :: p => Ups :: p
            case Ups :: Up :: p => Ups :: p
            case (c: Context) :: tail => c :: up2ups(tail)
            case x => x
        }
560
```

```
        /* id <: id.peer */
        var l: List[Univ] = new UPath(s, f, Peer :: c).simplify :: Nil
        if ( l.head == this )
            l = Nil

        /*
        *  id.peer <: id.up.rep
        *
        *  This follows from simplify(u) <: u but
        *  since I'm unsure how to implement it I'm
        *  doing this  explicitly . It's the most important
        *  case.
        *
        *  TODO
         */
        c match {
            case Peer :: p => l = new UPath(s, f, Up :: Rep :: p).simplify :: l
            case _ => ()
        }

        /* id.owner <: id.up */
        f.reverse match {
            case Owner :: tail => l = new UPath(s, f.init, Up :: c).simplify :: l
            case _ =>
        }

        /* p.rep <: p.reps && p.rep.reps <: p.reps */
        var cres = rep2reps(c.reverse).reverse
        if ( cres != c )
            l = new UPath(s, f, cres) :: l

        /* p.up <: p.ups && p.up.ups <: p.ups */
        cres = up2ups(c.reverse).reverse
        if ( cres != c )
            l = new UPath(s, f, cres) :: l

        /* p <: simplify(loosen(p)) */
        var tmp = this.loosen.simplify
        if ( tmp != this )
            l = tmp :: l

        /* eliminate duplicates (regain some efficiency) */
        l = l.removeDuplicates

        /* u <: u''  if  u <: u' && u' <: u'' */
        l.contains(path) || l.exists(_ <:< path) || {
            /* p.c <: p'.c  if  p <: p' */
            var t: List[Context] = Nil
            var c1 = c.reverse
            var c2 = path.c.reverse
            while ( (c1 != Nil) && (c2 != Nil) && (c1.head == c2.head) ) {
                t = c1.head :: t
                c1 = c1.tail
                c2 = c2.tail
            }
            (  t != Nil) && (new UPath(s, f, c1.reverse) <:<
                    new UPath(path.s, path.f, c2.reverse))
        }
```

```
    }
}


/**
 * Base class for start values/variables.
 *
 * @param id    the identifier of the value/variable.
 * @param _type the type of the value/variable.
 *
 * @author Daniel Schregenberger
 */
abstract class Start(id: String, _type: Type) {
    /** String representation (== id). */
    override def toString = id

    /** Getter for the type of the identifier. */
    def getType = _type

    /**
     * Construct a new path by selecting f on this.
     *
     * @param f the field to select.
     *
     * @return the generated path
     */
    def ° (f: Field) = new UPath(this, f :: Nil, Nil)

    /**
     * Construct a new path by selecting c on this.
     *
     * @param c the context to select.
     *
     * @return the generated path
     */
    def ° (c: Context) = new UPath(this, Nil, c :: Nil)

    /**
     * Construct a new path by selecting f on this.
     * Alias for °.
     *
     * @param f the field to select.
     *
     * @return the generated path
     */
    def + (f: Field) = this°f

    /**
     * Construct a new path by selecting c on this.
     * Alias for °.
     *
     * @param c the context to select.
     *
     * @return the generated path
     */
    def + (c: Context) = this°c
}
```

```
/**
 * Class representing local values.
 * Also used to represent method parameters and this.
 *
 * @param id    the identifier of the value.
 * @param _type the type of the value.
 *
 * @author Daniel Schregenberger
 */
case class Val (id: String, _type: Type) extends Start(id, _type)

/** Object representing the 'this' reference .. */
object This extends Val("this", null) {
    /** The type of this loops to itself . */
    override def getType = new NType(new UPath(this, Nil, Nil),
            new Class0("this"), new EmptyMap[TVarId, Type])
}

/**
 * Convenience object to create method parameters.
 * Applying Param actually creates a Val object but it
 * helps indicating that a value is supposed to be used
 * as method parameter.
 *
 * @author Daniel Schregenberger
 */
object Param {
    /**
     * Create value to be used as method parameter.
     *
     * @param id    the identifier of the parameter.
     * @param _type the type of the parameter.
     */
    def apply (id: String, _type: Type) = Val(id, _type)
}

/**
 * Class representing local variables.
 * Also used to represent method parameters and this.
 *
 * @param id    the identifier of the variables.
 * @param _type the type of the variables.
 *
 * @author Daniel Schregenberger
 */
final case class Var (id: String, _type: Type) extends Start(id, _type)

/**
 * Base class for fields.
 *
 * @param id       the identifier of the field.
 * @param _type    the type of the field.
 * @param _mutable whether the field is mutable or not.
 *
 * @author Daniel Schregenberger
 */
abstract class Field (id: String, _type: Type, _mutable: boolean) {
    /** String representation (== id). */
```

```
735     override def toString = id

        /** Getter for the type of the identifier. */
        def getType = _type

740     /** Getter for the mutable flag. */
        def isMutable = _mutable

        /**
         * Construct a new path by selecting f on this.
745      *
         * @param f the field to select.
         *
         * @return the new path
         *
750      * @throws error is the type of this field is a type variable.
         */
        def ° (f: Field) = _type match {
            case n: NType => This°this°f
            case _ => error("type variable dependency!")
755     }

        /**
         * Construct a new path by selecting c on this.
         *
760      * @param c the context to select.
         *
         * @return the new path
         *
         * @throws error is the type of this field is a type variable.
765      */
        def ° (c: Context): Univ = _type match {
            case n: NType => new UPath(This, this, c)
            case _ => error("type variable dependency!")
        }
770

        /**
         * Construct a new path by selecting f on this.
         * Alias for °.
         *
775      * @param f the field to select.
         *
         * @return the new path
         *
         * @throws error is the type of this field is a type variable.
780      */
        def + (f: Field): Univ = this°f

        /**
         * Construct a new path by selecting c on this.
785      * Alias for °.
         *
         * @param c the context to select.
         *
         * @return the new path
790      *
         * @throws error is the type of this field is a type variable.
         */
```

```
    def + (c: Context): Univ = this°c
}

/**
 * Class representing immutable fields.
 *
 * @param id    the identifier of the variables.
 * @param _type the type of the variables.
 *
 * @author Daniel Schregenberger
 */
final case class ValField (id: String, _type: Type) extends Field(id, _type, false)

/**
 * Class representing mutable fields.
 *
 * @param id    the identifier of the variables.
 * @param _type the type of the variables.
 *
 * @author Daniel Schregenberger
 */
final case class VarField (id: String, _type: Type) extends Field(id, _type, true)

/**
 * Object representing the (virtual) owner field.
 *
 * @param id    the identifier of the variables.
 * @param _type the type of the variables.
 *
 * @author Daniel Schregenberger
 */
object Owner extends Field("owner", new NType(This°Up, Object,
            new EmptyMap[TVarId, Type]), false)


/**
 * Base class for context modifiers.
 *
 * @param name the name of the context.
 *
 * @author Daniel Schregenberger
 */
abstract class Context (name: String) {
    /** String representation (== name). */
    override def toString = name
}

/** The peer context modifier. */
object Peer extends Context("peer")
/** The rep context modifier. */
object Rep extends Context("rep")
/** The up context modifier. */
object Up extends Context("up")
/** The reps context modifier. */
object Reps extends Context("reps")
/** The ups context modifier. */
object Ups extends Context("ups")
```

C.2 types.scala

```
package universes.formalizator;

import Coercions.Iterator2String
import scala.collection.immutable.EmptyMap

/**
 * Base class for types.
 *
 * @author Daniel Schregenberger
 */
abstract class Type {
    /**
     * Performs viewpoint adaptation of a type with regards to another type.
     * Also does the substitution of types for the corresponding type variables
     * Uses Univ.subst1 to perform viewpoint adaptatiosn of universe with
     * regards to other universes.
     *
     * @param viewpoints List of universe-value pair representing the
     *                   viewpoint(s).
     *
     * @return the viewpoint adapted type.
     *
     * @throws error if more than one value matches for the viewpoint
     *         adaptation of universes.
     * @see universes.formalizator.Univ.subst1
     */
    def subst (viewpoints: List[(Type, Val)]) : (Type, boolean)

    /**
     * Convert this type to a NType.
     * Also contains the environment mapping function Gamma for type variables:
     *     Type variables (TVarId) are replaced by their upper bound.
     * Use with care.
     *
     * @return the NType representing this type.
     */
    def toNType = this match {
        case n: NType => n
        case v: TVarId => {
            logger.warn("inserting upper bound " + v.uBound +
                    " for type variable " + v)
            v.uBound
        }
    }

    /**
     * The subtyping operator for types.
     *
     * @param t the type to test for being supertype of this.
     *
     * @return true if this <: t, false otherwise.
     */
    def <:<(t: Type) : boolean

    /**
     * Limited covariant subtyping for type arguments.
```

```
     *
     * @param t the type to test (with limited covariance)
     *          for being supertype of this.
     *
     * @return true if this <:_ arg t, false otherwise.
     */
    def arg_<:< (t: Type) : boolean
}

/**
 * Class representing type variables.
 *
 * @param id    the identifier of the variable.
 * @param cov   flag indicating whether the variable is covariant.
 * @param cont  flag indicating whether the variable is contravariant.
 * @param uBound fthe upper bound of the variable.
 *
 * @throws error if both cov and cont are true.
 */
final case class TVarId (id: String, cov: boolean, cont: boolean,
        uBound: NType) extends Type {
    if ( cov && cont )
        error("type variable " + id +
                " cannot be covariant and contravariante at once!")

    /**
     * String representation in Scala notation:
     * identifier prefixed by '+' or '-'.
     * depending on the variance of the variable.
     */
    override def toString = if ( cov ) "+" + id else if ( cont ) "-" + id else id

    /* see Type for documentation or use the generated ScalaDoc */
    def subst (viewpoints: List[(Type, Val)]) = {
        val res = viewpoints.filter { _._2 == This } match {
            case (n: NType, _ ) :: Nil =>
                    if ( n.typeParams.contains(this) ) {
                        n.typeParams(this) match {
                            case n: NType => (n, n.univ != Unknown)
                            case v => {
                                logger.warn("resolving type variable " + this +
                                        " to type variable " + v)
                                (    v, true)
                            }
                        }
                    }   else {
                        logger.warn("unresolved type variable " + this)
                        (   this , true)
                    }
            case _ => {
                logger.warn("unresolved type variable " + this)
              (   this , true)
            }
        }
        logger("subst(" + this + ", " +
                Iterator2String(viewpoints.elements) + ") = " + res)
        res
    }
```

```
115     /* see Type for documentation or use the generated ScalaDoc */
        def <:< (t: Type) = this == t

        /* see Type for documentation or use the generated ScalaDoc */
        def arg_<:< (t: Type) : boolean = t == this
120 }

    /**
     * Class representing NTypes, ie. parameterized types with main modifier.
     *
125  * @param univ       the main modifier for this type.
     * @param clazz      the (Scala) class of this type.
     * @param typeParams the type parameters for this type.
       *                  Since we do not have real types
       *                (  and a full featured real typesystem)
130    *                  it is a map of variables to types.
     */
    case class NType (univ: Univ, clazz: Class,
            typeParams: Map[TVarId, Type]) extends Type {
        /**
135      * String representation of the form: univ clazz[parameters]
         */
        override def toString = univ + " " + clazz.getName + {
            if ( ! typeParams.isEmpty )
              "[" + (  typeParams.values foldLeft "") {
140               (x, y) => x + {if (x != "") ", " + y else y}} + "]"
            else
              ""
        }

145     /**
         * Function to set the main modifier of a type.
         *
         * @param u the new main modifier.
         *
150      * @return new NType with main modifier u but the same class
          *         and type parameters as this one.
         */
        def setMain (u: Univ) = new NType(u, clazz, typeParams)

155     /**
         * Typing field selections.
         *
         * @param f the field to select on this as reciever.
         *
160      * @return the combined type for the selection.
         */
        def select (f: Field) = {
            logger("select(" + this + ", " + f + ", " + f.getType + ") = res").++(" | ")
            logger("calculating (Ttmp, btmp) ...").++
165         val (t1, bt) = f.getType.subst(List((this, This)))
            logger.--("(Ttmp, btmp) = " + (t1, bt))

            val res = univ match {
                case p: UPath => {
170                 if ( p.isStable && !f.isMutable ) {
                      (   t1.toNType.setMain(new UPath(p.s,
```

```
                                p.f ::: ( f :: Nil), Nil)), false )
            }   else
                (   t1, bt)
        }
          case _ => (t1, bt)
    }
     logger.--("res = " + res).--
     res
  }

  /* see Type for documentation or use the generated ScalaDoc */
  def subst (viewpoints: List[(Type, Val)]) = {
     logger("subst(" + this + ", " +
           Iterator2String(viewpoints.elements) + ") = res").++(" | ")
     logger("calculating (u1, b1) ...").++
     val (u1, b1) = univ.subst1(
           viewpoints.map(pi => (pi._1.toNType.univ, pi._2)))
     logger.--("(u1, b1) = " + (u1, b1))

     if ( ! typeParams.isEmpty )
        logger("calculating T2 ...").++
     val T2 = typeParams.transform((Xi, Ti) => Ti.subst(viewpoints))
     val Tr = T2.transform((Xi, Ti) => {
        if ( Ti._2 )
           Ti
        else {
           if ( Xi.cov )
              Ti
           else
            (   Ti._1.toNType.setMain(Unknown), false)
        }
     })
     val res = (new NType(u1, clazz, Tr.transform(
                   (Xi, Ti) => Ti._1)), b1 && Tr.values.forall(_._2))

     if ( ! typeParams.isEmpty ) {
        logger.--("T2 = " + Iterator2String(T2.values))
        logger("Tres = " + Iterator2String(Tr.values))
     }
     logger.--("res = " + res).--
     res
  }

  /* see Type for documentation or use the generated ScalaDoc */
  def <:<(t: Type) = t match {
     case n: NType => this <:< n
     case _ => false
  }

  /**
   * The subtyping operator for NTypes.
   *
   * @param t the NType to test for being supertype of this.
   *
   * @return true if this <: t, false otherwise.
   */
  def <:< (t: NType) : boolean = (univ <:< t.univ) &&
                (clazz <:< t.clazz) &&
```

```
230                 typeParams.forall(x => x match {
        case (v, p) => {
            val (l, r) = (p, t.typeParams(v))
            if ( v.cov )
              (  l <:< r)
            else if ( v.cont )
              (  r <:< l)
            else
               l arg_<:< r
        }
    })

    /* see Type for documentation or use the generated ScalaDoc */
    def arg_<:< (t: Type) : boolean = t match {
         case n: NType => this arg_<:< n
         case _ => false
    }

    /**
     * Limited covariant subtyping for type arguments.
     *
     * @param t the type to test (with limited covariance)
      *          for being supertype of this.
     *
     * @return true if this <:_arg t, false otherwise.
     */
    def arg_<:< (t: NType) : boolean = t.univ match {
        case Unknown => (clazz == t.clazz) &&
            typeParams.forall(x => x match {
                case (v, p) => p arg_<:< t.typeParams(v)
          })
        case u1 => (univ == u1) && (clazz == t.clazz) &&
            typeParams.forall(x => x match {
                case (v, p) => p arg_<:< t.typeParams(v)
          })
    }
}

/** The Unit type. Used for methods that do not return anything. */
object Unit extends NType(new UPath(Val("unit", null), Nil, Nil),
        new Class0("unit"), new EmptyMap[TVarId, Type]) {
    override def toString = "unit"
}

/** Object used to create classes. */
object Class extends Object with coercions {
    /**
     * Creates a new Class with no type parameters (Class0).
     *
     * @param name the name of the class.
     *
     * @return new Class0 object.
     */
    def apply (name: String) = new Class0(name)

    /**
     * Creates a new Class with one type parameter (Class1).
     *
```

```
     * @param name the name of the class.
     * @param a    the name of the first type variable (with
     *             variance annotation).
     *
     * @return new Class1 object.
     */
    def apply (name: String, a: String) = new Class1(name, a)

    /**
     * Creates a new Class with two type parameters (Class2).
     *
     * @param name the name of the class.
     * @param a    the name of the first type variable (with
     *             variance annotation).
     * @param b    the name of the second type variable (with
     *             variance annotation).
     *
     * @return new Class2 object.
     */
    def apply (name: String, a: String, b: String) = new Class2(name, a, b)
}

/**
 * Abstract class representing classes.
 * We are using different subclasses depending on the number of
 * type parameters for actual classes. This has the benefit that a
 * compile time error is raised if a wrong number of type arguments
 * is specified.
 *
 * @param name the name of the class.
 */
abstract class Class (name: String) {
    /**
     * Reference to the superclass.
     * Defaults to this since it is impossible to reference the Object
     * class as it inherits from this class.
     * But it is set to Object for all other classes.
     */
    var superclass = this

    /** Getter for the name. */
    def getName = name

    override def toString = getName

    /**
     * The subclassing operator.
     * We only support subclassing for non-parameterized types so far.
     *
     * @param c the class to test for being superclass of this.
     *
     * @return true if this == c.
     */
    def <:< (c: Class) = (this == c)
}

/**
 * Class representing classes with no type parameters.
```

```
 *
 * @param name the name of the class.
 */
class Class0 (name: String) extends Class(name) {
  /** Create the instance once and cache it. */
  val instance = new Instance(this)

  superclass = Object

  /**
   * The extends clause for our system.
   * We only support subclassing for non-parameterized types so far.
   *
   * @param c the instance of the super class.
   *
   * @return this with updated superclass.
   */
  def _extends (c: Class0) = {
    superclass = c
    this
  }

  /**
   * Get the Instance of this class.
   *
   * @return the Instance of this class.
   */
  def apply() = instance

  /**
   * The subclassing operator for non-parameterized types.
   *
   * @param c the class to test for being superclass of this.
   *
   * @return true if this is a subclass of c.
   */
  override def <:< (c: Class) = (c == this) || (c == Object) ||
                                (superclass <:< c)
}

/**
 * Class representing classes with one type parameter.
 *
 * @param name the name of the class.
 * @param a    the first type variable.
 */
final class Class1 (name: String, a: TVarId) extends Class(name) {
  /**
   * Creates a new Instance of this class by binding the type parameters.
   *
   * @param Ta the Type for the first type variable.
   *
   * @return new Instance of this class.
   */
  def apply (Ta: Type) = new Instance(this, Map(a -> Ta))
}

/**
```

```
   * Class representing classes with two type parameter.
405 *
   * @param name the name of the class.
   * @param a    the first type variable.
   * @param b    the second type variable.
   */
410 final class Class2 (name: String, a: TVarId, b: TVarId) extends Class(name) {
      /**
       * Creates a new Instance of this class by binding the type parameters.
       *
       * @param Ta the Type for the first type variable.
415    * @param Tb the Type for the second type variable.
       *
       * @return new Instance of this class.
       */
      def apply (Ta: Type, Tb: Type) = new Instance(this, Map(a -> Ta, b -> Tb))
420 }

   /**
    * This class represents classes that have all their type parameters bound
    * to some Type. It is only needed as intermediate result before it is
425 * combined with a universe to form a full NType.
    * Having this intermediate type makes sure no type arguments are forgotten
    * when adding the ownership modifier (universe).
    *
    * @param clazz      the (Scala) class for this instance.
430 * @param typeParams the type parameters for this instance.
    *
    * @see Univ.##
    * @see NType
    */
435 case class Instance (clazz: Class, typeParams: Map[TVarId, Type]) {
      /**
       * Alternative constructor for type-parameterless classes
       *
       * @param clazz      the (Scala) class for this instance.
440    */
      def this (clazz: Class) = this(clazz, new EmptyMap[TVarId, Type])
   }

   /**
445 * This object is our representation of the root class for
    * all classes (ie. java.lang.Object).
    */
   object Object extends Class("Object") {
      /** Create the instance once and cache it. */
450   val instance = new Instance(this)

      /**
       * Get the Instance of this class.
       *
455    * @return the Instance of this class.
       */
      def apply() = instance
   }
```

C.3 misc.scala

```
0  package universes.formalizator;

   /**
    * This trait contains various coercions which help to to write
    * expressions for our system in a more natural way and reduce
5   * the typing overhead for programers.
    * Just mix it into classes using the system.
    */
   trait coercions {
     /**
10    * Convert start value to NType.
      * This is essentially the environment mapping function Gamma.
      *
      * @param s the start value to map.
      *
15    * @return the NType for the start value.
      */
     implicit def Start2NType (s: Start): NType = s match {
       case x: Var => x.getType.toNType
       case a: Val => new NType(new UPath(a, Nil, Nil),
20                     a.getType.toNType.clazz, a.getType.toNType.typeParams)
     }

     /**
      * Convert a String to a type variable.
25    * Accepts variance annotations (T, +T, -T) and returns
      * the type variable with the respective variance.
      * Also allows to just write "T" when a type variable is
      * used as type for a field for example.
      *
30    * @param x the String to convert.
      *
      * @return type variable with the given variance.
      */
     implicit def String2TVarId (x: String): TVarId = x(0) match {
35     case '+' =>
         new TVarId(x.substring(1, x.length), true, false , Any ## Object)
       case '-' =>
         new TVarId(x.substring(1, x.length), false, true , Any ## Object)
       case _ => new TVarId(x, false, false , Any ## Object)
40   }

     /**
      * Converts a context to a UPath by prefixing it with This.
      * Allows to use the shorter form Peer instead of This°Peer.
45    *
      * @param c the context to convert.
      *
      * @return a path consisting of start value this and context c.
      */
50   implicit def Context2Univ (c: Context) = new UPath(This, Nil, c :: Nil)

     /**
      * Get the instance for a class with no type arguments.
      * For convenience, without this function one would have
55    * to write C() to get the instance of class C.
```

```
    *
    * @param c the class to convert.
    *
    * @return instance of this class.
    */
   implicit def Class02Instance(c: Class0) = c.instance

   /**
    * Get the instance for the Object class.
    * For convenience, without this function one would have
    * to write Object() to get the instance.
    *
    * @param o reference to the Object singleton object
    *
    * @return instance of Object.
    */
   implicit def Object2Instance(o: Object.type) = Object.instance

   /**
    * Get the invocation for a method with no arguments.
    * For convenience, without this function one would have
    * to write m() to get the invocation of a parameterless
    * method m.
    *
    * @param m the method to convert.
    *
    * @return instance of this class.
    */
   implicit def Method02Invocation(m: Method0) = m.invocation

   /**
    * Provides a compact String representation for Lists etc.
    * The default representation for Lists "List(1, 2, 3)" may be
    * confusing since we have examples with classes named "List".
    *
    * @param i the collection to convert.
    *
    * @return compact representation
    */
   def Iterator2String[A](i: Iterator[A]) = "{" + ("" /: i)(
         (x, y) => x + {if( x != "" ) ", " + y else y}) + "}"
}

/**
 * Object used to allow importing only selected coercions functions.
 */
object Coercions extends coercions

/** Object used to create methods. */
object Method {
   /**
    * Creates a new method with no parameters.
    *
    * @param name the name of the class.
    *
    * @return new method.
    */
   def apply(name: String, ret: Type) = new Method0(name, ret)
```

```
115     /**
        * Creates a new method with one parameter.
        *
        * @param name the name of the class.
        * @param p1   the first parameter.
120     *
        * @return new method.
        */
       def apply (name: String, p1: Val, ret: Type) =
           new Method1(name, p1 :: Nil, ret)
125
       /**
        * Creates a new method with two parameters.
        *
        * @param name the name of the class.
130     * @param p1   the first parameter.
        * @param p2   the second parameter.
        *
        * @return new method.
        */
135    def apply (name: String, p1: Val, p2: Val, ret: Type) =
           new Method2(name, p1 :: p2 :: Nil, ret)

       /**
        * Creates a new method with three parameters.
140     *
        * @param name the name of the class.
        * @param p1   the first parameter.
        * @param p2   the second parameter.
        * @param p2   the third parameter.
145     *
        * @return new method.
        */
       def apply (name: String, p1: Val, p2: Val, p3: Val, ret: Type) =
           new Method3(name, p1 :: p2 :: p3 :: Nil, ret)
150 }

   /**
    * Abstract class representing methods (method signatures).
    * We are using different subclasses depending on the number of
155 * parameters. This has the benefit that a compile time error is
    * raised if a wrong number of arguments is specified.
    *
    * @param name   the name of the method.
    * @param params the parameter list.
160 * @param ret    the type of the return value. Use Unit if the method
      *              has no return value.
    */
   abstract class Method (name: String, params: List[Val], ret: Type) {
       /**
165     * Returns the method signature: def name (p1: T1, p2: T2): ret
        *
        * @return the method signature.
        */
       override def toString = "def " + name + (params foldLeft "(") {
170        (x, y) => x + {if (x != "(") ", "  else ""} + y + ": " + y.getType } +
           "): " + ret
```

```
}

/**
 * Class representing methods with no parameters.
 *
 * @param name the name of the method.
 * @param ret  the type of the return value. Use Unit if the method
 *             has no return value.
 */
class Method0 (name: String, ret: Type) extends Method(name, Nil, ret) {
    /** Create the invocation once and cache it. */
    val invocation = new MethodInvocation(name, Nil, ret)

    /**
     * Get the Invocation of this method.
     *
     * @return the Invocation of this method.
     */
    def apply() = invocation
}

/**
 * Class representing methods with one parameter.
 *
 * @param name the name of the method.
 * @param params the parameter list.
 * @param ret    the type of the return value. Use Unit if the method
 *               has no return value.
 */
class Method1 (name: String, params: List[Val], ret: Type) extends
        Method(name, params, ret) {
    /**
     * Create an Invocation of this method.
     *
     * @param Ta the type of the first argument.
     *
     * @return the Invocation of this method.
     */
    def apply(Ta: Type) = new MethodInvocation(name, (Ta :: Nil) zip params, ret)
}

/**
 * Class representing methods with two parameters.
 *
 * @param name the name of the method.
 * @param params the parameter list.
 * @param ret    the type of the return value. Use Unit if the method
 *               has no return value.
 */
class Method2 (name: String, params: List[Val], ret: Type) extends
        Method(name, params, ret) {
    /**
     * Create an Invocation of this method.
     *
     * @param Ta the type of the first argument.
     * @param Tb the type of the second argument.
     *
     * @return the Invocation of this method.
```

```
230       */
      def apply(Ta: Type, Tb: Type) =
          new MethodInvocation(name, (Ta :: Tb :: Nil) zip params, ret)
  }

235 /**
   * Class representing methods with three parameters.
   *
   * @param name the name of the method.
   * @param params the parameter list.
240 * @param ret    the type of the return value. Use Unit if the method
    *              has no return value.
   */
  class Method3 (name: String, params: List[Val], ret: Type) extends
          Method(name, params, ret) {
245     /**
       * Create an Invocation of this method.
       *
       * @param Ta the type of the first argument.
       * @param Tb the type of the second argument.
250    * @param Tc the type of the third argument.
       *
       * @return the Invocation of this method.
       */
      def apply(Ta: Type, Tb: Type, Tc: Type) =
255       new MethodInvocation(name, (Ta :: Tb :: Tc :: Nil) zip params, ret)
  }

  /**
   * This class represents method invocations.
260 * In other words it represents the call to a method where
   * all argument types are specified.
   *
   * @param name   the name of the method.
   * @param args   the argument/parameter list.
265 * @param ret    the type of the return value. Use Unit if the method
    *              has no return value.
   */
  case class MethodInvocation (name: String, args: List[(Type, Val)], ret: Type) {
      /**
270    * Returns a "signature" of the method call: name (T1, T2)
       *
       * @return the invocation signature.
       */
      override def toString = name + (args foldLeft "(") {
275       (x, y) => x + {if (x != "(") ", "  else ""} + y._1} + ")"
  }
```

C.4 logger.scala

```
0 package universes.formalizator;

  import scala.util.logging._

  /**
5  * A simple logger object used to collect and print (or suppress)
   * debug messages, log messages and warnings.
   *
   * Debug messages are intended to be used to debug the implementation.
   * Log messages are usefull to monitor the process of typing a certain
10 * expression.
   * Warnings are used to warn about lossy simplifications of paths and
   * similar things.
   */
  object logger extends logger {
15   /** Flag indicating whether debug messages are printed. Default: false. */
     private var doDebug = false
     /** Flag indicating whether log messages are printed. Default: false. */
     private var doLog = false
     /** Flag indicating whether warnings are printed. Default: true. */
20   private var doWarn = true

     /**
      * Flag used to reset the logger when printing somewhere else
      * (testcase progress indication for example).
25    * The next message printed by the logger will be preceded by a newline
      * to make sure the output does not look too messed up.
      */
     private var tainted = false

30   /** Toggle the debug flag. */
     def toggleDebugging () = doDebug = !doDebug
     /** Toggle the log flag. */
     def toggleLogging () = doLog = !doLog
     /** Toggle the warning flag. */
35   def toggleWarnings () = doWarn = !doWarn

     /** Set the tainted flag. */
     def taint () = tainted = true

40   /**
      * Takes care of cleaning the output if tainted.
      * (Prints a newline if tainted.)
      */
     def init () = {
45       if ( tainted )
             Console.println
         tainted = false
     }

50   override def debug(m: => String) = {
         if ( doDebug ) {
             init()
             Console.println("DEBUG: " + m)
        }
55   }
```

```
    override def log(m: => String) = {
        if ( doLog ) {
            init()
            Console.println(m)
        }
    }
    override def warn(m: => String) = {
        if ( doWarn ) {
            init()
            Console.println("WARNING: " + m)
        }
    }
}

/**
 * Basic functionality of the logger.
 */
trait logger {
    /** Indentation prefix for log messages. */
    private var n = ""

    /**
     * Increase indentation by two spaces.
     *
     * @return this which allows to nest calls to the logger.
     */
    def ++ : logger = ++("  ")

    /**
     * Increase indentation by a String.
     *
     * @param s indentation string to add.
     *
     * @return this which allows to nest calls to the logger.
     */
    def ++ (s: String) = { n += s; this }

    /**
     * Decrease indentation by two characters.
     *
     * @return this which allows to nest calls to the logger.
     */
    def -- = { n = n.substring(0, n.length - 2); this }

    /**
     * Function used to print debug messages.
     *
     * @param m parameterless function evaluating to String.
     *          Only evaluated if the message is really printed.
     */
    def debug (m: => String) = {}

    /**
     * Function used to print log messages.
     *
     * @param m parameterless function evaluating to String.
     *          Only evaluated if the message is really printed.
     */
```

```
    def log (m: => String) = {}

    /**
     * Function used to print warnings.
     *
     * @param m parameterless function evaluating to String.
     *          Only evaluated if the message is really printed.
     */
    def warn (m: => String) = {}

    /**
     * Applying the logger will print the message prefixed by the
     * indentation String n.
     *
     * @param m log message.
     *
     * @return this which allows to nest calls to the logger.
     */
    def apply (m: String) = { log(n + m); this }
}
```

Bibliography

[1] Jonathan Aldrich and Craig Chambers. Ownership domains: Separating aliasing policy from mechanism. In Martin Odersky, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086, pages 1–25. Springer-Verlag, 2004.

[2] Chris Andreae, James Noble, Shane Markstrum, and Todd Millstein. A framework for implementing pluggable type systems. *SIGPLAN Not.*, 41(10):57–74, October 2006.

[3] Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership types for object encapsulation. In *ACM Symposium on Principles of Programming Languages (POPL)*, New Orleans, Louisiana, January 2003.

[4] Dave Clarke and Sophia Drossopoulou. Ownership, Encapsulation and the Disjointness of Types and Effects. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'02)*, pages 292–310, Seattle, Washington, USA, November 2002. ACM Press.

[5] Dave Clarke, Sophia Drossopoulou, James Noble, and Tobias Wrigstad. Tribe: a simple virtual class calculus. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*, pages 121–134, New York, NY, USA, 2007. ACM Press.

[6] Vincent Cremet, François Garillot, Sergueï Lenglet, and Martin Odersky. A core calculus for Scala type checking. In *Proceedings of MFCS*, Springer LNCS, September 2006.

[7] Dave Cunningham, Sophia Drossopoulou, Susan Eisenbach, Werner Dietl, and Peter Müller. Universes for race safety. January 2007.

[8] Werner Dietl, Sophia Drossopoulou, and Peter Müller. Generic Universe Types. In Erik Ernst, editor, *European Conference on Object-Oriented Programming (ECOOP)*, Lecture Notes in Computer Science. Springer-Verlag, 2007. To appear.

[9] Werner Dietl and Peter Müller. Exceptions in ownership type systems. In Erik Poll, editor, *Formal Techniques for Java-like Programs*, pages 49–54, 2004.

[10] Werner Dietl and Peter Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology (JOT)*, 4(8):5–32, October 2005.

[11] Burak Emir, Martin Odersky, and John Williams. Matching Objects with Patterns. Technical report, 2006.

[12] Erik Ernst. Family polymorphism. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 303–326, Budapest, Hungary, 2001.

[13] Erik Ernst. Higher-order hierarchies. In Luca Cardelli, editor, *Proceedings ECOOP 2003*, LNCS 2743, pages 303–329, Heidelberg, Germany, July 2003. Springer-Verlag.

[14] Erik Ernst, Klaus Ostermann, and William R. Cook. A virtual class calculus. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 270–282, New York, NY, USA, 2006. ACM Press.

[15] Atsushi Igarashi and Mirko Viroli. Variant path types for scalable extensibility. In *FOOL/-WOOD '07 — International Workshop on Foundations and Developments of Object-Oriented Languages, Nice, France Proceedings*, January 2007.

[16] Stefan Nägeli. Ownership in design patterns. Master's thesis, March 2006.

[17] Martin Odersky. Scala by example.
`http://www.scala-lang.org/docu/files/ScalaByExample.pdf`, June 2007.

[18] Martin Odersky. The Scala Language Specification, version 2.4.
`http://www.scala-lang.org/docu/files/ScalaReference.pdf`, February 2007.

[19] Martin Odersky and al. An overview of the scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[20] Martin Odersky, Vincent Cremet, Iulian Dragos, Gilles Dubochet, Burak Emir, Sean McDirmid, Stéphane Micheloud, Nikolay Mihaylov, Lex Spoon, and Matthias Zenger. An introduction to Scala.
`http://scala.epfl.ch/docu/files/ScalaIntro.pdf`, July 2006.

[21] Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A Nominal Theory of Objects with Dependent Types. In *Proc. ECOOP 2003*, Springer LNCS 2743, July 2003.

[22] Martin Odersky, Christoph Zenger, and Matthias Zenger. Colored Local Type Inference. In *POPL*, 2001.

[23] Jan Schäfer and Arnd Poetzsch-Heffter. Simple loose ownership domains. Technical Report 348/06, Department of Computer Science, University of Kaiserslautern, March 2006.

[24] Michel Schinz. A Scala tutorial for Java programmers.
`http://www.scala-lang.org/docu/files/ScalaTutorial.pdf`, June 2007.

[25] Daniel Schregenberger. Runtime checks for the Universe type system, 2004. Semester thesis.

[26] Sun Microsystems, Inc. The Java language specification.
`http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html`.

[27] Yoshimi Takano. Implementing uniqueness and ownership transfer in the Universe type system. Master's thesis, March 2007.