Implementing a Universe Type System for Scala

Manfred Stock

Master Thesis Project Report

Software Component Technology Group
Department of Computer Science
ETH Zurich

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

August 2007 – January 2008

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

The Universe type system is based on the concept of ownership where every object has at most one owner object. It structures the object store using different contexts, which are sets of objects with the same owner, and restricts how references can be passed and used. When enforcing the owner-as-modifier property, the Universe type system guarantees that any modification of an object can only be initiated by its owner object.

Scala is a multi-paradigm programming language combining features of object-oriented and functional languages. It supports compiler plugins and annotations on types which allows the implementation of additional type constraints such as those imposed by the Universe type system.

This thesis presents an implementation of the Universe type system for Scala. The implementation supports a subset of the Scala language. It provides two compiler plugins for the Scala compiler and a set of annotations for the use in programs. One plugin performs the static Universe type checks and the other plugin inserts additional runtime checks during compilation.

Acknowledgements

I would like to thank my supervisor Werner Dietl for his support and feedback. Particular thanks go to Lex Spoon for his help with certain aspects of Scala and especially for his work on type annotations.

Special thanks also go to my parents and to my sister for their support during my studies and the work on this master thesis.

Contents

1 Introduction

This first chapter gives short introductions to the Scala programming language and to the type rules of the Universe type system. The introduction to Scala highlights some of its differences to Java and especially those features which are relevant for this thesis. This comprises its compiler plugins which were used to implement a Universe type system for Scala, Scala's type inference, and its support for annotations on types.

1.1 Scala

Scala [20] is a programming language which combines functional and object-oriented features. It is compiled to Java byte code and is therefore interoperable with Java. There are many tutorials for Scala, for example [25, 34, 16]. The next few subsections nevertheless give a short introduction to Scala, especially to those features which were used in the implementation of a Universe type system presented in this thesis.

Every value in Scala is an object, there are no primitive types like in Java [19, Section 3]. Scala supports a similar set of object-oriented features like Java, but there are some differences:

- There is usually no explicit constructor method, but instead the class body itself serves as one. References and methods defined in the class body become the fields and methods of the class. Parameters to the default constructor are therefore specified when defining the class as can be seen for `class Complex` in Listing 1.1. It is nevertheless possible to overload the constructor method by defining other `def this(<parameters>)` methods with different parameters.

  A noteworthy feature of the default constructor is that parameters marked with `var` or `val` are automatically transformed into mutable or final fields of the class, respectively.

- A Scala class may not have static members. Instead, Scala supports the creation of *singleton objects* by using the keyword `object`. Such an object cannot be instantiated explicitly, but it gets created automatically the first time it is used. The *singleton* in the name already implies that there is only one instance of an object in the scope of its definition and below.

  Static members of a class can be simulated by using a special so-called *companion object* which shares its name with a class. Other objects are also called *stand-alone objects*. It is important to distinguish between these two kinds of objects because companion objects get more privileges, such as access to private members of the class they share the name with.

- Scala does not use interfaces, but a similar concept called *traits*. Traits can be seen as a construct between interfaces and abstract classes: They may contain method signatures and implementations of methods like an abstract class. In addition, other classes can extend several traits, which is similar to Java's handling of interfaces which are therefore internally mapped to Scala's traits. This so-called *mixin-class composition* allows the combination of several classes containing implementations, but without the complications of multiple inheritance.

A function in Scala is also a value which allows the implementation of higher-order functions such as `map`. In addition, there are several other features shared between Scala and most common functional programming languages. This includes nesting of function definitions and modelling of algebraic types through its `case` classes and its built-in pattern matching.

```
package complex
/** Static object for complex numbers */
object Complex {
  val i = Complex(0,1)
  implicit def double2complex(x: Double) = Complex(x,0)
  implicit def int2complex(x: Int) = Complex(x.toDouble,0)
  def apply(re: Double, im: Double) = new Complex(re,im)
}
/** Class representing complex numbers */
class Complex(val re: Double, val im: Double) {
  def +(that: Complex) = Complex(this.re + that.re, this.im + that.im)
  def -(that: Complex) = Complex(this.re - that.re, this.im - that.im)
  def *(that: Complex) = Complex(this.re * that.re - this.im * that.im,
                                 this.re * that.im + this.im * that.re)
  def /(that: Complex) = {
    val denom = that.re * that.re + that.im * that.im
    Complex(
        (this.re * that.re + this.im * that.im) / denom,
        (this.im * that.re - this.re * that.im) / denom
    )
  }
  override def toString = re+(if (im < 0) "-" + (-im) else"+" + im) + "*I"
}
```

Listing 1.1: Implementing complex numbers for Scala.

Listing 1.1 contains an example of a Scala program which was taken from [17]: It implements complex numbers as a library. One can see several syntactical similarities to Java but also some differences, three of which seem particularly prominent: (1) Semicolons are mostly optional, they are only required if one wants to put more than one statement on a line. (2) Types of variables, the method return type and type arguments usually do not need to be specified: Scala is able to infer them, except for recursive functions and definitions. (3) Names of methods are less restricted than in Java, it is therefore possible to declare operators like `+`, `-`, etc. as methods of a class. However, one needs to be aware of the operator precedence and associativity which are derived from the first and last character, respectively, of the method's name [18, Section 6.12.3].

1.1.1 Implicit Conversions and Views

After importing the `Complex` singleton object's members using `import Complex._` into the current scope, it is possible to seamlessly work with complex numbers: `1 + 1 * i` for example is a legal expression now. The reason why this works are *implicit conversions* called *views*. There are two cases where the Scala compiler applies views:

1. If an expression `e` is of type `T`, and `T` does not conform to the expression's expected type `T'`.
2. If the selector `m` does not denote a member of type `T` in a selection `e.m` where `e` has type `T`.

In the first case, a view `v`, i.e. a function declared as `implicit`, is searched which is applicable to `e` and whose result type conforms to `T'`. In the other case, a view `v` is searched which is applicable to `e` and whose result provides a member conforming to `m`.

The expression `1 + 1 * i` is processed as follows: First the compiler handles `1 * i` which could be rewritten as `1.*(i)`. Since `i` is of type `Complex` and `1` is an `Int` instance, the compiler cannot find a matching method `*(that: Complex)` in class `Int`. It therefore searches for a view which converts an `Int` to a type providing such a method. As `object Complex` provided an implicit conversion from `Int` to `Complex` with the function `int2complex`, the compiler will implicitly apply this view. The same will be done for the subsequent addition.

1.1.2 Class Hierarchy

Scala uses a unified object model with `scala.Any` at the base of its class hierarchy. See Figure 1.1 for a diagram showing this hierarchy.

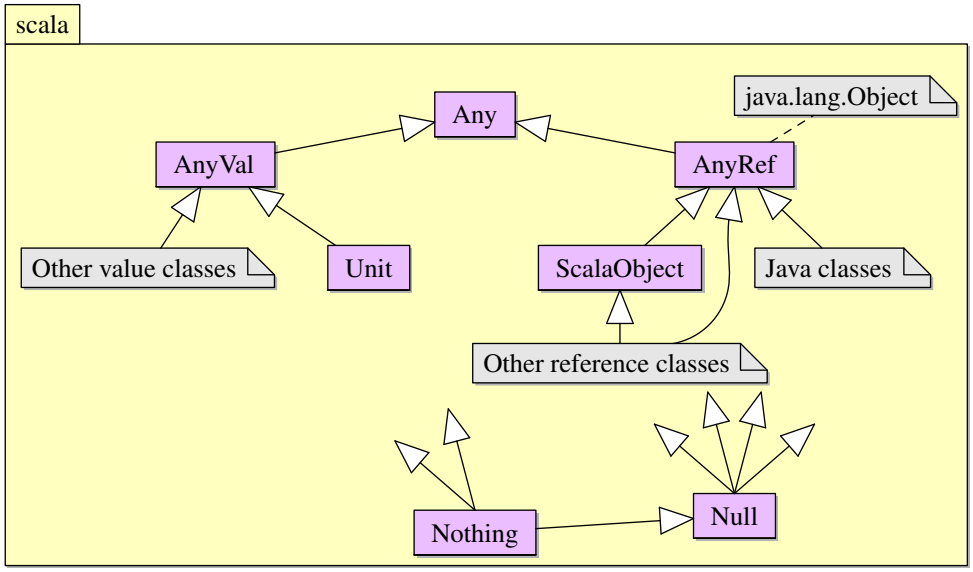


Figure 1.1: Class hierarchy of Scala.

There are two categories for the subclasses of `scala.Any`: *Value classes* subclassing `scala.AnyVal` and *reference classes* subclassing `scala.AnyRef`. Each primitive type of Java has a corresponding value class while `scala.AnyRef` is identified with `java.lang.Object`. `scala.Unit` basically corresponds to Java's `void` but there actually is an instance of `scala.Unit`, namely the value `()`.

The two types at the bottom of the class hierarchy do not have a direct analogon in Java: `scala.Null` is a subtype of all reference types and its only instance is the `null` reference. This prevents the assignment of `null` to an instance of a value class since `scala.Null` is not a subtype of `scala.AnyVal`. The type `scala.Nothing` is a subtype of every other type. There exists no instance of `scala.Nothing` but it is useful as a type parameter's lower bound.

Arrays

There is no special syntax for arrays in Scala. Instead, it uses the normal, parameterized class `Array[A]` to represent polymorphic arrays. It is nevertheless possible to mimic the syntax of Java by using a more general syntactic concept of Scala: When applying parentheses to a variable while passing in some arguments, Scala will transform this into an invocation of a method named `apply` if possible. This can be seen in Listing 1.1: `object Complex` provides an `apply` method which takes two arguments. It is therefore allowed in this case to write `Complex(0,1)` in order to create a new instance of a complex number.

For arrays, this is similar: Let `a` be of type `Array[Int](10)`. `a(5)` would then return the element at index position `5`. There is also an akin concept for updates: An expression like `a(5) = 10` gets mapped to `a.update(5,10)` if `a`'s type provides a conforming `update` method, which is the case for `Array[A]`.

First-Class Functions

The Scala compiler desugars first-class functions to concrete implementations of the `FunctionN` trait, with $N \in 0 \ldots 9$. Parameters of the first-class function are turned into parameters of the

specialization's `apply` function and the body of the first-class function becomes the body of the `apply` function.

In addition, it is also necessary to adapt higher-order functions as their actual arguments are not functions, but instances of `FunctionN`. Their parameter types are therefore transformed such that they subsequently take `FunctionN` instances. The body of a higher-order function gets adapted, too: Applications of the first-class function are turned into calls of the `apply` method of the respective `FunctionN` instance.

The above transformations make is possible to handle first-class functions just like any other object. Hence, first class functions also fit into Scala's unified object model, and it is usually not necessary to treat them specifically.

1.1.3 Type Inference

According to [5, Section 5] the type inference in Scala is based on *Colored Local Type Inference* [21] which is a refinement of local type inference [24]. Local type inference basically means that the inference uses only information from adjacent nodes in the syntax tree to recover missing type information. This avoids global constraint solving and implies that the type is inferred from the initialization of a variable or the definition of a method.

1.1.4 Annotations

Scala supports annotations on definitions and declarations, types, and expressions. An annotation has the form `@a` or `@a(a1,a2,...)`. In these cases, `a` is the constructor of a class conforming to `scala.Annotation`. Annotations directly extending `scala.Annotation` are neither preserved for the Scala type checker nor for the class file. They are only visible locally during the compilation run which analyzes them. However, specializations of the alternate `scala.StaticAnnotation` trait which extends `scala.Annotation` are available to Scala's type checker in every compilation unit where the annotated construct is accessed.

1.1.5 Self types

Self-type annotations allow one to attach a type to `this` which is different from the surrounding class. Combined with Scala's abstract type members and its modular mixin composition, this results in a mechanism for the construction of reusable components [22]. This mechanism is employed at the core of the implementation of the Scala compiler and also in the implementation of the Universe type system presented in this thesis.

The self type of a class or object must conform to the self types of all classes it inherits from. Listing 1.2 shows an example of a class `A` whose self type was declared as being `B`. It is now possible to write the expression `(new A with B).saySomething` which results in "Hello, B-World!" being printed. Similarly, one could write the expression `(new A with C).saySomething` which would result in "Hello, C-World!". Of course one may also declare a named class like in `class D extends A with C` and get the same result as in the second example when running `(new D).saySomething`.

1.1.6 Compiler Plugins

The Scala compiler may be extended by writing compiler plugins. These plugins are also written in Scala and get access to the abstract syntax tree of the standard compiler after a specified compiler phase. A list of available phases may be obtained by executing the command `scalac -Xshow-phases`. The output of this command depends on the plugins which are currently in use.

Since compiler plugins get access to the complete abstract syntax tree, they may conduct additional checks or even modify it to change the runtime behavior of the program. Both variants are used in this thesis in order to execute static type checks on the programs and to add runtime checks to the generated byte code (see Sections 4.3 and 4.4).

```
class A {
  self: B =>
  def saySomething = sayHello
}

trait B {
  def sayHello = println("Hello, B-World!")
}

trait C extends B {
  override def sayHello = println("Hello, C-World!")
}
```

Listing 1.2: Example of using Scala's self types.

There are two abstract classes provided by the Scala compiler which must be extended when implementing a compiler plugin. See Figure 1.2 for the overall structure of these classes. The `Plugin` class is the entry point to a compiler plugin. It contains the name and a list of components which are provided by the plugin and which should be executed during the compilation. A `PluginComponent` must specify the name of the compiler phase after which it should be run. See Listing 1.3 for a basic example of a "Hello, World!" compiler plugin which implements these abstract classes. It prints a message and the filename of each source file it processes.

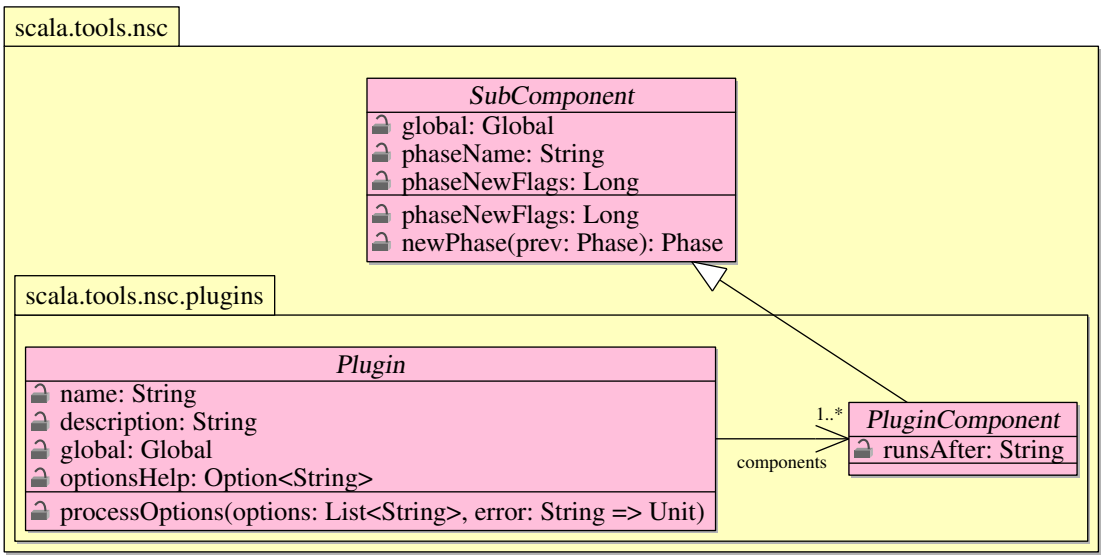


Figure 1.2: Basic hierarchy for compiler plugins.

A `Plugin` gets a `global` reference to an instance of class `scala.tools.nsc.Global` at runtime. This class provides many methods and inner classes required during compilation as well as access to the settings of the current compilation run. The provided classes comprise both those for the representation of the abstract syntax tree and those for Scala's type description. The field `global` of class `SubComponent` is abstract, hence actual `PluginComponent` implementations must implement this field by providing a `val global: Global` class parameter. This ensures that the `PluginComponent` gets access to the same settings and classes as the `Plugin` and all other compiler phases.

The `runsAfter` field of the `PluginComponent` is used to tell the compiler after which phase the component should run. Both built-in phases and phases defined in other plugins are allowed.

Plugins are packaged as Java archive files which contain two files besides the compiled classes: (1) A file named `scalac-plugin.xml` (see Listing 1.4) containing the name of the plugin and the

```
package ch.ethz.inf.sct.uts.plugin.helloworld
import scala.tools.nsc._
import scala.tools.nsc.plugins._

/** Compiler plugin for the Scala compiler. */
class HelloWorldPlugin(val global: Global) extends Plugin {
  /** Name of this plugin. */
  val name = "helloworld"

  /** Description of this plugin. */
  val description = "Compiler plugin which prints out 'Hello, World!'"

  /** Components of this plugin. */
  val components = List(new HelloWorldPluginComponent(global))

  /** Handling of any plugin-specific options. */
  override def processOptions(options: List[String],
      error: String => Unit) {
    options foreach {
      case o => error("Unknown option "+o+" for "+name)
    }
  }

  /** A description of this plugin's options, for -help. */
  override val optionsHelp: Option[String] = None
}

/** Plugin component which runs after runsAfter. */
class HelloWorldPluginComponent (val global: Global)
  extends PluginComponent {
  import global._
  /** Name of this compiler phase. */
  val phaseName = "helloworld-phase"

  /** When to execute this phase. */
  val runsAfter = "refchecks"

  /** Factory to create the new phase for the graph generation. */
  def newPhase(prev: Phase) = new Phase(prev) {
    def name = phaseName

    /** Process the compilation units. */
    def run {
      println("Hello, World!")
      currentRun.units foreach {u => println("Processing "+u.source.path)}
      println("Done.")
    }
  }
}
```

Listing 1.3: Basic "Hello, World!" compiler plugin.

```
<?xml version="1.0" encoding="utf-8"?>
<plugin>
    <name>
      helloworld
    </name>
    <classname>
      ch.ethz.inf.sct.uts.plugin.helloworld.HelloWorldPlugin
    </classname>
</plugin>
```

Listing 1.4: Example for the `scalac-plugin.xml` file.

```
version.major=0
version.minor=1
```

Listing 1.5: Example for the `helloworld.version.properties` file.

name of the class implementing the abstract `Plugin` class. (2) An optional file providing some version information for the plugin named `<plugin-name>.version.properties` (see Listing 1.5). The version information is currently not used, but one could employ it during the compilation of the plugin. A future version of the Scala compiler might also make use of this information if the compiler gets support for the specification of explicit dependencies between plugins, which is currently not possible.

Using Compiler Plugins

In order to use a plugin, one needs to pass the option `-Xplugin:<path-to-plugin.jar>` to the `scalac` command. This results in the execution of the plugin's components after the phases specified in the field `runsAfter` of the `PluginComponent` instances. It is possible to specify more than one plugin or a directory with several plugins. The latter can be done using `-Xpluginsdir <dir>` which will load and use all plugins from `dir` during compilation.

There are two more options that might be useful when working with plugins: (1) The option `-Xplugin-list` prints the list of active plugins together with their synopsis. This is for example useful if some plugins have been copied to the special directory mentioned in Section 2.2.2: Plugins from this directory are loaded automatically and may therefore change the normal compilation process. If such a plugin contains an error, it may modify the compilation in a detrimental way – and this option is a facility to identify such spurious plugins. (2) The more general option `-Xshow-phases` displays the order in which the phases of built-in compiler components and the phases from plugins are executed during compilation. One case is particularly interesting: It may happen that there are two or more plugins which define new phases that should be executed after the same existing phase. If for some reason the sequence of execution for these new phases must be changed, one may simply load the plugins in a different order.

1.1.7 Universe Types in Scala

The objective of this thesis is that the user should be able to write down a Scala program like the one in Listing 1.6. It includes only a few so-called *ownership modifiers* represented as source code annotations – the missing ownership modifiers should be inferred or defaulted automatically by the compiler plugins. After inferring these modifiers, the type rules of the Universe type system (see Section 1.2.1) should be checked as well. If the user used type casts or instanceof tests in the program, additional checks must be done at runtime [26]. These should also be added during compilation.

```
package ch.ethz.inf.sct.uts.examples
import ch.ethz.inf.sct.uts.annotation._
class A {
  val b = new (B @rep)
  val s = b.s
  def t = new (Cls @peer)
  val u = b.t
}
```

Listing 1.6: Simple example of the usage.

1.2 Universe Type System

The Universe type system [15, 7, 6] is used to control aliasing and dependencies in object-oriented programs by hierarchically structuring the object store. Its underlying basis is the concept of ownership where each object is owned by at most one distinct owner object.

All objects of a program execution are organised into *contexts*, which are sets of objects with the same owner. Objects without owner are grouped into the so-called *root context*, which also forms the root of the tree of contexts of a program execution. When enforcing the so-called *owner-as-modifier* discipline, the owner must have control over the modification of its objects.

Ownership modifiers statically express object ownership relative to some object called the *viewpoint*. The ownership modifiers of the Universe type system are always relative to `this`: A `peer` reference denotes a reference to an object in the same context as `this` while a `rep` reference identifies an object as being owned by `this`. An `any` reference may cross context boundaries and reference an object with statically unknown owner.

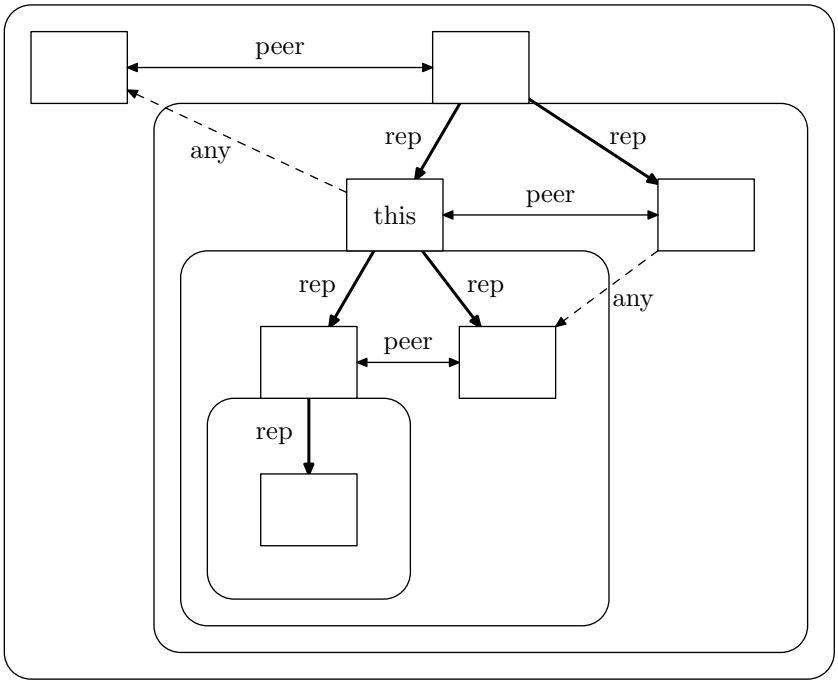


Figure 1.3: Ownership relations in an object structure.

Figure 1.3 shows an exemplar object store. The arrows denote references between objects which are annotated with ownership modifiers. Dashed arrows are used for `any` references which may cross context boundaries while `rep` references are expressed by slightly thicker arrows. Rectangles

with rounded corners define contexts whereas the outermost rectangle is the root context.

The following subsection gives a formal introduction to the actual type rules which are implemented in this thesis. It can safely be skipped if one is not interested in the rules: Chapter 2 presents a more informal overview of the type system which should suffice in order to understand the remainder of this thesis.

1.2.1 Type Rules

This subsection introduces the variant of the Universe type system's rules which is implemented in this thesis. These rules are based on a revised version of the type system presented in [6] and have in addition been adapted to Scala. The explanations are therefore basically a summary of those found in [6], but adapted to the changed rules.

Since the presented rules are taken from an intermediate version between [6] and as yet unpublished work, there is no soundness proof for the type system. The original work in [6] also discusses a runtime model which is not covered in this thesis. Instead, the runtime checks are based on earlier work for Java [26] and do therefore not support generics. Such support was implemented for Java in [23] and could be implemented for Scala as well. The focus of this thesis is also on the implementation of a Universe type system checker for Scala, and not on the refinement of the type system. However, the implementation provides a framework which makes it easy to experiment with different variants of the Universe type system for Scala. It also provides a default implementation which uses the type rules of this subsection.

Syntax and Type Environments

Figure 1.4 presents the syntax of the language and the naming conventions for variables. Both are based on and therefore similar to [6], but they have been adapted to match the syntax of Scala: Type parameters and arguments are therefore written in squared brackets. Method signatures contain the `def` keyword, the type parameters are between the method's name and its parameters, and the return type follows the parameters, separated by a colon. The ownership modifier also follows the type, which is certainly the most significant difference. Type casts are written differently as Scala implements them via calls to the parameterized `asInstanceOf[T]: T` method.

A possibly empty sequence of `T`s is denoted by $\overline{\texttt{T}}$. The i-th element of the sequence is identified by $\texttt{T}_i$, and the empty sequence is denoted by ϵ. A sequence of tuples $\overline{\texttt{X T}}$ constitutes a map. Substitutions are written with double brackets like in $[\![\texttt{new/old}]\!]$ because the meaning of single squared brackets could be ambiguous in the presence of type parameters. In addition, support for the *if-then-else* expression has been added to the rules. However, the rules do not model constructor methods nor interfaces and traits which are nevertheless handled by the implementation.

P	∈	Program	::=	$\overline{\texttt{Cls}}$ C e
Cls	∈	Class	::=	class C[$\overline{\texttt{X N}}$] extends C[$\overline{\texttt{T}}$] { $\overline{\texttt{f T}}$; $\overline{\texttt{mt}}$ }
T	∈	Type	::=	N | X
N	∈	NType	::=	C[$\overline{\texttt{T}}$] u
u	∈	OM	::=	$\texttt{peer}_u$ | $\texttt{rep}_u$ | $\texttt{some}_u$ | $\texttt{any}_u$ | $\texttt{this}_u$ | $\texttt{lost}_u$
mt	∈	Meth	::=	w def m[$\overline{\texttt{X N}}$]($\overline{\texttt{x T}}$): T = { $\overline{\texttt{e}}$ }
w	∈	Purity	::=	pure | nonpure
e	∈	Expr	::=	null | x | e.f | e.f=e | e.m[$\overline{\texttt{T}}$]($\overline{\texttt{e}}$) | new N | e.asInstanceOf[T] | if ($\texttt{e}_0$) {$\texttt{e}_1$} else {$\texttt{e}_2$}
Γ	∈	Env	::=	$\overline{\texttt{X N}}$; $\overline{\texttt{x T}}$

Figure 1.4: Syntax and type environments.

A program P ∈ Program consists of any number of classes and a main class with a main expression e ∈ Expr. Classes Cls ∈ Class may have type parameters and they can specialize

another class. Their body may define methods and fields.

The type rules distinguish two kinds of types: A non-variable $\texttt{NType}$ and type variables $\texttt{X} \in \texttt{TVarID}$. Non-variable types may have any number of type arguments $\texttt{T} \in \texttt{Type}$ and a main ownership modifier $\texttt{u} \in \texttt{OM}$. The subscript u is omitted if it is clear from context that an ownership modifier is meant.

There are some restrictions on the use of ownership modifiers. `this` and `lost` are internal modifiers, they cannot be used in the program's source code. `this` is only employed for the type of the `this` reference while `lost` indicates that ownership information got lost during the viewpoint adaptation. In addition, `some` is only allowed in type arguments, but not as a main modifier.

Methods $\texttt{mt} \in \texttt{Meth}$ have a purity indicator $\texttt{w} \in \texttt{Purity}$, any number of type and method parameters, and a method return type. Their body contains any number of expressions $\texttt{e} \in \texttt{Expr}$.

Type checks take place in a type environment $\Gamma \in \texttt{Env}$. It maps type variables to their upper bounds and method parameters to their types. The notation $\Gamma(\texttt{X})$ refers to the upper bound of type variable $\texttt{X}$, while $\Gamma(\texttt{x})$ refers to the type of method parameter $\texttt{x}$.

Viewpoint Adaptation

All ownership modifiers express ownership relative to an object, the so-called *viewpoint*. Since the type rules use `this` as the viewpoint, all types $\texttt{T}$ which are seen from another type $\texttt{T}'$ need to be adapted to the viewpoint `this`. This so-called *viewpoint adaptation* is done by the overloaded $\triangleright$ operator: It adapts (1) an ownership modifier from a viewpoint described by another ownership modifier, (2) a type from a viewpoint described by an ownership modifier, and (3) a type from a viewpoint described by another type.

Adapting an Ownership Modifier w.r.t. an Ownership Modifier The viewpoint adaptation of an ownership modifier from another ownership modifier could be explained using a field access `e.f`: Let $\texttt{u}$ be the main ownership modifier of `e` and $\texttt{u}'$ be the main ownership modifier of `e`'s field `f`. Then relative to `this`, the type of the field access `e.f` has the main modifier $\texttt{u} \triangleright \texttt{u}'$.

$$
\begin{array}{rcll}
\triangleright :: \texttt{OM} \times \texttt{OM} & \rightarrow & \texttt{OM} & \\
\texttt{peer} \triangleright \texttt{peer} & = & \texttt{peer} & \\
\texttt{u} \triangleright \texttt{some} & = & \texttt{some} & \\
\texttt{this} \triangleright \texttt{u}' & = & \texttt{u}' & \text{if } \texttt{u}' \neq \texttt{this}
\end{array}
\qquad
\begin{array}{rcll}
\texttt{rep} \triangleright \texttt{peer} & = & \texttt{rep} & \\
\texttt{u} \triangleright \texttt{any} & = & \texttt{any} & \\
\texttt{u} \triangleright \texttt{u}' & = & \texttt{lost} & \text{otherwise}
\end{array}
$$

Accessing the reference `f` to a `peer` object through a `peer` reference yields `peer` since all involved objects have the same owner. If $\texttt{u}$ is `rep`, that is `this` is the owner of `e`, and $\texttt{u}'$ is `peer`, then a `peer` object of `e` is also owned by `this`, which results in `rep`.

Adapting `some` from the viewpoint of an arbitrary ownership modifier to the viewpoint `this` results in `some`. In the case that $\texttt{u}'$ is `any`, `f` already points to an object in an unknown context. Accessing the object through `e` does not result in more precise ownership information, hence this yields `any`.

As `this` is only used for the type of `this`, no viewpoint adaptation is necessary because the viewpoint already is `this`. In all other cases, ownership information gets lost during the viewpoint adaptation, hence the resulting modifier is `lost`.

Adapting a Type w.r.t. an Ownership Modifier Type variables are not subject to viewpoint adaptation because the substitution by actual type arguments takes place in the scope where the type variables are instantiated. The type arguments of non-variable types are adapted recursively.

$$
\begin{array}{rcl}
\triangleright :: \texttt{OM} \times \texttt{Type} & \rightarrow & \texttt{Type} \\
\texttt{u} \triangleright \texttt{X} & = & \texttt{X} \\
\texttt{u} \triangleright \texttt{C}[\,\overline{\texttt{T}}\,]\ \texttt{u}' & = & \texttt{C}[\,\overline{\texttt{u} \triangleright \texttt{T}}\,]\ (\texttt{u} \triangleright \texttt{u}')
\end{array}
$$

Adapting a Type w.r.t. a Type A type T gets adapted from the viewpoint described by a non-variable type:

$$\begin{array}{rcl} \triangleright :: \texttt{NType} \times \texttt{Type} & \rightarrow & \texttt{Type} \\ \texttt{C}[\overline{\texttt{T}}]\ \texttt{u}\triangleright\texttt{T} & = & (\texttt{u}\triangleright\texttt{T})[\![\overline{\texttt{T}[\![\texttt{lost/some}]\!]/\texttt{X}}]\!] \qquad \text{where } \overline{\texttt{X}}{=}\text{dom}(\texttt{C}) \end{array}$$

The ownership modifiers of T are adapted first, then its type arguments get processed: Type arguments $\overline{\texttt{T}}$ are substituted for the type variables $\overline{\texttt{X}}$ of C. dom(C) denotes C's type variables $\overline{\texttt{X}}$ in the declaration class C$[\overline{\texttt{X}\ _}]$ ... of class C. The substitution of some by lost is done in order to express the fragility of some.

Subclassing

The term *subclassing* is used to describe the relation on classes as declared by the extends keyword – it does not take ownership modifiers into account. *Subtyping* on the other hand respects ownership modifiers.

$$\text{SC-1}\ \frac{\texttt{class C}[\overline{\texttt{X}\ \_}]\ \texttt{extends C}'[\overline{\texttt{T}'}]}{\texttt{C}[\overline{\texttt{X}}] \sqsubseteq \texttt{C}'[\overline{\texttt{T}'}]} \qquad \text{SC-2}\ \frac{}{\texttt{C}[\overline{\texttt{X}}] \sqsubseteq \texttt{C}[\overline{\texttt{X}}]}$$

$$\text{SC-3}\ \frac{\texttt{C}[\overline{\texttt{X}}] \sqsubseteq \texttt{C}''[\overline{\texttt{T}''}] \quad \texttt{C}''[\overline{\texttt{X}''}] \sqsubseteq \texttt{C}'[\overline{\texttt{T}'}]}{\texttt{C}[\overline{\texttt{X}}] \sqsubseteq \texttt{C}'[\overline{\texttt{T}'[\![\texttt{T}''/\texttt{X}'']\!]}]}$$

Figure 1.5: Rules for subclassing.

The subclass relation $\sqsubseteq$ is defined on instantiated classes C$[\overline{\texttt{T}}]$. Its rules in Figure 1.5 state the following: Each un-instantiated class is a subclass of the class it extends (SC-1). Subclassing is reflexive (SC-2) and transitive (SC-3).

Subuniversing

There is also a relation between ownership modifiers, similar to a simple subtype relation, called *subuniversing*. It is shown in Figure 1.6.

$$\text{SU-1}\ \frac{}{\texttt{this}_u <:_\texttt{u} \texttt{peer}} \qquad \text{SU-2}\ \frac{}{\texttt{peer} <:_\texttt{u} \texttt{lost}}$$

$$\text{SU-3}\ \frac{}{\texttt{rep} <:_\texttt{u} \texttt{lost}} \qquad \text{SU-4}\ \frac{}{\texttt{lost} <:_\texttt{u} \texttt{some}}$$

$$\text{SU-5}\ \frac{}{\texttt{some} <:_\texttt{u} \texttt{any}} \qquad \text{SU-6}\ \frac{}{\texttt{u} <:_\texttt{u} \texttt{u}}$$

Figure 1.6: Rules for subuniversing.

This relation is used in subtyping rule ST-2 in order to express that e.g. C rep is a subtype of C any. The relation builds a tree with any at the root.

Subtyping and Limited Covariance

The subtype relation $<:$ of Figure 1.7 is defined on types and $\Gamma \vdash \texttt{T} <: \texttt{T}'$ expresses that T is a subtype of $\texttt{T}'$ in the type environment Γ.

Rule ST-1 conveys that two types sharing the same ownership modifier are subtypes if the corresponding classes are subclasses. However, ownership modifiers in $\texttt{T}'$ are relative to the instance

$$\text{ST-1}\frac{\mathtt{C}[\,\overline{\mathtt{X}}\,] \sqsubseteq \mathtt{C}'[\,\overline{\mathtt{T}'}\,]}{\Gamma \vdash \mathtt{C}[\,\overline{\mathtt{T}}\,]\ \mathtt{u} <: \mathtt{C}'[\,\overline{(\mathtt{u} \triangleright \mathtt{T}')[\![\mathtt{T}/\mathtt{X}]\!]}\,]\ \mathtt{u}} \qquad \text{ST-2}\frac{\mathtt{u} <:_{\mathtt{u}} \mathtt{u}' \quad \Gamma \vdash \overline{\mathtt{T} <:_{\mathtt{a}} \mathtt{T}'}}{\Gamma \vdash \mathtt{C}[\,\overline{\mathtt{T}}\,]\ \mathtt{u} <: \mathtt{C}[\,\overline{\mathtt{T}'}\,]\ \mathtt{u}'}$$

$$\text{ST-3}\frac{\Gamma \vdash \mathtt{T} <: \mathtt{T}'' \quad \Gamma \vdash \mathtt{T}'' <: \mathtt{T}'}{\Gamma \vdash \mathtt{T} <: \mathtt{T}'} \qquad \text{ST-4}\frac{\mathtt{T} = \mathtt{X} \vee \Gamma(\mathtt{X}) <: \mathtt{T}}{\Gamma \vdash \mathtt{X} <: \mathtt{T}}$$

$$\text{TA-1}\frac{}{\Gamma \vdash \mathtt{T} <:_{\mathtt{a}} \mathtt{T}} \qquad \text{TA-2}\frac{\mathtt{u}' \in \{\mathtt{u}, \mathtt{some}\} \quad \Gamma \vdash \overline{\mathtt{T} <:_{\mathtt{a}} \mathtt{T}'}}{\Gamma \vdash \mathtt{C}[\,\overline{\mathtt{T}}\,]\ \mathtt{u} <:_{\mathtt{a}} \mathtt{C}[\,\overline{\mathtt{T}'}\,]\ \mathtt{u}'}$$

$$\text{TA-3}\frac{\mathtt{T} = \Gamma(\mathtt{X})[\![\mathtt{some}/\mathtt{peer}, \mathtt{some}/\mathtt{rep}, \mathtt{some}/\mathtt{any}]\!]}{\Gamma \vdash \mathtt{X} <:_{\mathtt{a}} \mathtt{T}}$$

Figure 1.7: Rules for subtyping and limited covariance.

of class `C`, therefore a viewpoint adaptation and a substitution of type variables needs to take place. Two types are also subtypes if their corresponding classes are the same, their main ownership modifiers fulfill the subuniversing relation, and if their type arguments satisfy the covariant subtyping relation $<:_{\mathtt{a}}$ (ST-2). The subtyping relation is also transitive (ST-3) and reflexivity of $<:$ follows from ST-2. A type variable `X` is a subtype of type `T` if `T` is `X` itself or if the upper bound of `X` is a subtype of `T` (ST-4).

Covariant subtyping is reflexive (TA-1). Two types are covariant subtypes if the corresponding classes are the same, if their main ownership modifiers are either equal or if the prospective supertype's main modifier is `some`, and if their type arguments are also covariant subtypes (TA-2). Finally, TA-3 expresses that `X` is a covariant subtype of `T` if `T` equals the upper bound of `X` where the `peer`, `rep` and `any` ownership modifiers were substituted by `some`.

Variances Scala supports variance annotations of type parameters of generic classes. It is therefore possible to declare type parameters as being covariant or contravariant. However, this is neither supported in the above subtyping rules nor the implementation, although earlier work [27] already provided adapted rules.

Lookup Functions

The lookup functions defined below are used in the type rules to get the possibly viewpoint adapted type of a field or signature of a method.

Field Lookup $\mathit{fType}(\mathtt{C}, \mathtt{f})$ returns the type of field `f` in class `C` or an undefined result if class `C` does not declare field `f`. If the function is called with a non-variable type `N` instead of a class, the result will be viewpoint adapted.

$$\text{SFT-1}\frac{\mathtt{class\ C}[\,\_\,]\ \mathtt{extends}\ \_[\,\_\,]\ \{\ \ldots \mathtt{T\ f} \ldots;\ \_\ \}}{\mathit{fType}(\mathtt{C}, \mathtt{f}) = \mathtt{T}}$$

$$\text{SFT-2}\frac{\mathtt{N} = \mathtt{C}[\,\_\,]\ \_ \quad \mathit{fType}(\mathtt{C}, \mathtt{f}) = \mathtt{T}}{\mathit{fType}(\mathtt{N}, \mathtt{f}) = \mathtt{N} \triangleright \mathtt{T}}$$

Method Lookup As for the above *fType* function there are two variants of the *mType* function: One which returns the method signature as it is declared in the defining class and one which adapts the viewpoint of the method signature's types before returning the signature. If the given method `m` is not declared in class `C`, the result is undefined.

$$\text{SMT-1}\ \frac{\texttt{class C[\_] extends \_[\_] \{ \_; \ldots w def m[}\overline{\texttt{X}_m\ \texttt{N}_b}\texttt{]}(\overline{\texttt{x T}_p})\texttt{: T}_r \ldots \}}{mType(\texttt{C},\texttt{m}) = \texttt{w def m[}\overline{\texttt{X}_m\ \texttt{N}_b}\texttt{]}(\overline{\texttt{x T}_p})\texttt{: T}_r}$$

$$\text{SMT-2}\ \frac{\begin{array}{c}\texttt{N} = \texttt{C[\_]}\ \_ \qquad mType(\texttt{C},\texttt{m}) = \texttt{w def m[}\overline{\texttt{X}_m\ \texttt{N}_b}\texttt{]}(\overline{\texttt{x T}_p})\texttt{: T}_r \\ \texttt{N} \triangleright \texttt{N}_b = \texttt{N}'_b \qquad \texttt{N} \triangleright \texttt{T}_r = \texttt{T}'_r \qquad \texttt{N} \triangleright \texttt{T}_p = \texttt{T}'_p\end{array}}{mType(\texttt{N},\texttt{m}) = \texttt{w def m[}\overline{\texttt{X}_m\ \texttt{N}'_b}\texttt{]}(\overline{\texttt{x T}'_p})\texttt{: T}'_r}$$

Overloaded methods are not explicitly supported. However, if the actual method lookup is not solely based on the name, they can be handled as well. The original version of the type system in [6] did not allow overloaded methods, but as Scala already relies on overloading for simple arithmetic expressions like $0 + 1$, this had to be relaxed.

Well-Formedness

Types, methods, classes, programs and type environments must be well-formed. Enforcement of the rules summarized in Figure 1.8 guarantees the well-formedness property for these constructs.

$$\text{WFT-1}\ \frac{\texttt{X} \in \text{dom}(\Gamma)}{\Gamma \vdash \texttt{X ok}} \qquad \text{WFT-2}\ \frac{\texttt{class C[}\overline{\_\ \texttt{N}}\texttt{]}\ \ldots \quad \Gamma \vdash \overline{\texttt{T}}\ \texttt{ok} \quad \Gamma \vdash \overline{\texttt{T}} <: ((\texttt{C[}\overline{\texttt{T}}\texttt{]}\ \texttt{u}) \triangleright \overline{\texttt{N}})}{\Gamma \vdash \texttt{C[}\overline{\texttt{T}}\texttt{]}\ \texttt{u ok}}$$

$$\text{WFM-1}\ \frac{\begin{array}{c}\Gamma = \overline{\texttt{X}_m\ \texttt{N}_b}, \overline{\texttt{X N}}; \texttt{this}\ (\texttt{C[}\overline{\texttt{X}}\texttt{]}\ \texttt{this}_u), \overline{\texttt{x T}_p} \\ \Gamma \vdash \texttt{T}_r, \overline{\texttt{N}_b}, \overline{\texttt{T}_p}\ \texttt{ok} \quad \Gamma \vdash \overline{\texttt{e}} : \overline{\texttt{T}} \quad \Gamma \vdash \texttt{e}_0 : \texttt{T}_r \quad override(\texttt{C},\texttt{m})\end{array}}{\texttt{w def m[}\overline{\texttt{X}_m\ \texttt{N}_b}\texttt{]}(\overline{\texttt{x T}_p})\texttt{: T}_r = \{\ \overline{\texttt{e};}\ \texttt{e}_0\ \}\ \texttt{ok in C[}\overline{\texttt{X N}}\texttt{]}}$$

$$\text{WFM-2}\ \frac{\begin{array}{c}\forall\ \texttt{class C}'\texttt{[}\overline{\texttt{X}'\ \texttt{N}'}\texttt{]}: \quad \texttt{C[}\overline{\texttt{X}}\texttt{]} \sqsubseteq \texttt{C}'\texttt{[}\overline{\texttt{T}'}\texttt{]} \wedge \text{dom}(\texttt{C}) = \overline{\texttt{X}} \Rightarrow \\ mType(\texttt{C}',\texttt{m}) \text{ is undefined } \vee\ mType(\texttt{C},\texttt{m}) = mType(\texttt{C}',\texttt{m})[\![\overline{\texttt{T}'/\texttt{X}'}]\!]\end{array}}{override(\texttt{C},\texttt{m})}$$

$$\text{WFC}\ \frac{\begin{array}{c}\overline{\texttt{X N}}; \_ \vdash \overline{\texttt{N}}, \overline{\texttt{T}}, (\texttt{C}'\texttt{[}\overline{\texttt{T}'}\texttt{]}\ \texttt{this}_u)\ \texttt{ok} \\ \overline{\texttt{mt}}\ \texttt{ok in C[}\overline{\texttt{X N}}\texttt{]} \qquad \texttt{rep} \notin \overline{\texttt{N}}\end{array}}{\texttt{class C[}\overline{\texttt{X N}}\texttt{]}\ \texttt{extends C}'\texttt{[}\overline{\texttt{T}'}\texttt{]}\ \{\ \overline{\texttt{mt};}\ \overline{\texttt{f T}}\ \}\ \texttt{ok}}$$

$$\text{WFP}\ \frac{\begin{array}{c}\overline{\texttt{Cls}}\ \texttt{ok} \\ \texttt{class C[ ]} \ldots \in \overline{\texttt{Cls}} \\ \epsilon;\ \texttt{this}\ (\texttt{C[ ]}\ \texttt{this}_u) \vdash \texttt{e} : \texttt{N}\end{array}}{\overline{\texttt{Cls}}, \texttt{C}, \texttt{e ok}} \qquad \text{SWFE}\ \frac{\begin{array}{c}\Gamma = \overline{\texttt{X N}}, \overline{\texttt{X}'\ \texttt{N}'}\ ;\ \texttt{this}\ (\texttt{C[}\overline{\texttt{X}}\texttt{]}\ \texttt{this}_u), \overline{\texttt{x T}} \\ \texttt{class C[}\overline{\texttt{X N}}\texttt{]}\ \ldots \qquad \Gamma \vdash \overline{\texttt{N}}, \overline{\texttt{N}'}, \overline{\texttt{T}}\ \texttt{ok}\end{array}}{\Gamma\ \texttt{ok}}$$

Figure 1.8: Well-formedness rules.

The judgment $\Gamma \vdash \texttt{T ok}$ means that type T is well-formed in type environment Γ. A type variable is well-formed if its upper bound is contained in Γ (WFT-1). Non-variable types are well-formed if their type arguments are well-formed and if the type arguments are subtypes of the viewpoint adapted upper bound of the corresponding type parameters (WFT-2).

The judgment $\texttt{mt ok in C[}\overline{\texttt{X N}}\texttt{]}$ expresses that method mt is well-formed in a class C with type parameters $\overline{\texttt{X N}}$. According to WFM-1, this is the case if: (1) The return type, the upper bounds of type parameters and the parameter types are well-formed, (2) the type of the method body conforms to the declared return type, and (3) the method correctly overrides any matching method in its superclasses (WFM-2). Overriding rules are respected if no method is overridden at all or

if the overriding method has the same signature as the overridden method after substituting type variables in the superclass' method.

Actual implementations of the type system may safely relax the requirement for equality of method signatures as follows: It suffices if the return type of the overriding method is a subtype of the overridden method's return type (covariance). In addition, the overriding method's parameter types may be supertypes of the overridden method's parameter types (contravariance). Overriding a non-pure method with a pure method would also be safe.

The judgment `Cls ok` expresses that class declaration `Cls` is well-formed. Rule WFC conveys that this is the case if (1) the upper bounds of `Cls`'s type variables, the types of its fields, and the instantiation of the superclass are well-formed; (2) `Cls`'s methods are well-formed; (3) no upper bound contains the `rep` modifier.

The judgment `P ok` expresses that program `P` is well-formed. According to rule WFP this applies if all classes in `P` are well-formed, the main class `C` is non-generic, and the main expression `e` is well-typed.

The judgment Γ `ok` expresses that type environment Γ is well-formed. For this to hold, rule SWFE requires that all upper bounds and method parameters are well formed. In addition, `this` must be mapped to a non-variable type with main modifier `this` and an uninstantiated class.

Type Rules

Figure 1.9 presents the actual type rules. The judgment $\Gamma \vdash$ `e : T` expresses that expression `e` is well-typed with type `T` in environment Γ. Type rules can only be applied if all involved types are well-formed in the respective environment.

$$\text{GT-Subs}\ \frac{\Gamma \vdash \mathtt{e} : \mathtt{T} \qquad \Gamma \vdash \mathtt{T} <: \mathtt{T}'}{\Gamma \vdash \mathtt{e} : \mathtt{T}'} \qquad \text{GT-Var}\ \frac{\mathtt{x} \in \mathrm{dom}(\Gamma)}{\Gamma \vdash \mathtt{x} : \Gamma(\mathtt{x})} \qquad \text{GT-Null}\ \frac{\mathtt{T} \neq \_[\,\_\,]\ \mathtt{this}_u}{\Gamma \vdash \mathtt{null} : \mathtt{T}}$$

$$\text{GT-New}\ \frac{\mathtt{N} \neq \_[\,\_\,]\ \mathtt{any}_u \qquad \mathtt{some}, \mathtt{lost} \notin \mathtt{N}}{\Gamma \vdash \mathtt{new\ N} : \mathtt{N}} \qquad \text{GT-Cast}\ \frac{\Gamma \vdash \mathtt{e}_0 : \mathtt{T}_0}{\Gamma \vdash \mathtt{e}_0.\mathtt{asInstanceOf}[\,\mathtt{T}\,] : \mathtt{T}}$$

$$\text{GT-If}\ \frac{\Gamma \vdash \mathtt{e}_0 : \mathtt{Boolean} \quad \Gamma \vdash \mathtt{e}_1 : \mathtt{T}_1 \quad \Gamma \vdash \mathtt{e}_2 : \mathtt{T}_2 \qquad \Gamma \vdash \mathtt{T}_1 <: \mathtt{T} \qquad \Gamma \vdash \mathtt{T}_2 <: \mathtt{T}}{\Gamma \vdash \mathtt{if}\ (\mathtt{e}_0)\ \{\mathtt{e}_1\}\ \mathtt{else}\ \{\mathtt{e}_2\} : \mathtt{T}}$$

$$\text{GT-Read}\ \frac{\Gamma \vdash \mathtt{e}_0 : \mathtt{N}_0 \qquad \mathit{fType}(\mathtt{N}_0, \mathtt{f}) = \mathtt{T}}{\Gamma \vdash \mathtt{e}_0.\mathtt{f} : \mathtt{T}} \qquad \text{GT-Upd}\ \frac{\Gamma \vdash \mathtt{e}_0 : \mathtt{N}_0 \quad \Gamma \vdash \mathtt{e}_2 : \mathtt{T} \qquad \mathit{fType}(\mathtt{N}_0, \mathtt{f}) = \mathtt{T} \quad \mathtt{lost} \notin \mathtt{T} \qquad \boxed{\mathtt{N}_0 = \_[\,\_\,]\ \mathtt{u} \qquad \mathtt{u} \in \{\mathtt{peer}, \mathtt{rep}, \mathtt{this}\}}^{OaM}}{\Gamma \vdash \mathtt{e}_0.\mathtt{f}{=}\mathtt{e}_2 : \mathtt{T}}$$

$$\text{GT-Invk}\ \frac{\begin{array}{c}\Gamma \vdash \mathtt{e}_0 : \mathtt{N}_0 \\ \mathit{mType}(\mathtt{N}_0, \mathtt{m}) = \mathtt{w\ def\ m}[\,\overline{\mathtt{X}_m\ \mathtt{N}_b}\,](\overline{\mathtt{x\ T}_p}) : \mathtt{T}_r \qquad \mathtt{lost} \notin \mathtt{T}_p \circ \mathtt{N}_b \\ \Gamma \vdash \overline{\mathtt{T}} <: \overline{\mathtt{N}_b}[\![\,\overline{\mathtt{T}/\mathtt{X}_m}\,]\!] \qquad \Gamma \vdash \overline{\mathtt{e}_2} : \overline{\mathtt{T}_p}[\![\,\overline{\mathtt{T}/\mathtt{X}_m}\,]\!] \\ \boxed{\mathtt{N}_0 = \_[\,\_\,]\ \mathtt{u} \qquad \mathtt{u} \notin \{\mathtt{peer}, \mathtt{rep}, \mathtt{this}\} \Rightarrow \mathtt{w} = \mathtt{pure}}^{OaM}\end{array}}{\Gamma \vdash \mathtt{e}_0.\mathtt{m}[\,\overline{\mathtt{T}}\,](\overline{\mathtt{e}_2}) : \mathtt{T}_r[\![\,\overline{\mathtt{T}/\mathtt{X}_m}\,]\!]}$$

Figure 1.9: Type rules.

An expression of type `T` can also be typed with a supertype `T'` of `T` (GT-Subs). The type

of method parameters (including the implicit `this` parameter) can be retrieved from the type environment (GT-Var). The `null` reference may have any ownership modifier except for `this` (GT-Null). New instances of a class must be created in a specific context, therefore `any` is not allowed (GT-New). The type of a cast is the type the object gets casted to (GT-Cast). An if-then-else expression is correctly typed if the type of both branches is a subtype of the expression's type (GT-If).

The type of a field read is the type of the field adapted from the receiver's type (GT-Read). In the case that the receiver's type is a type variable, its upper bound is used. A field update is similar (GT-Upd). However, it is not allowed to update a field whose type contains `lost` or to update it with a value of a non-conforming type. If the owner-as-modifier property is being enforced, then `this` must be the owner or a peer of the receiver, or the receiver itself.

Finally, rule GT-Invk handles method calls, which are similar to field reads and updates: The upper bounds of type arguments and the method arguments' types must not contain `lost`. Types of method and type arguments must be subtypes of the method parameter types and the type parameter's upper bounds, respectively, with substituted type variables. The type of the expression is the method's return type from its signature where type variables have been substituted. If the owner-as-modifier property should be enforced, only pure methods may be called on receivers which point to an unknown context.

Separating Encapsulation and Topology In deviation from the original type rules the variant used in this thesis is divided into two parts, as indicated above: It distinguishes between (1) the ownership topology and (2) aliasing control. The basic rules only establish the ownership topology. In order to enforce encapsulation, the type rules from Figure 1.9 are extended by additional rules marked with a frame and *OaM*. These rules enforce the owner-as-modifier property.

Default Ownership Modifiers

According to [6], earlier work in [7] has shown that the annotation overhead can be reduced by using appropriate defaults. In general, the default ownership modifier is `peer`. For upper bounds of type variables, exceptions, and immutable types it defaults to `any`. Immutable types are for example `Int` and `String`.

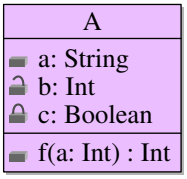
1.3 Notation and Naming Conventions

Usually the term *type annotation* refers to the type of a variable, method return type, etc. which is written down in the source code, for example `Int` in `val a: Int = 0`. Scala also supports annotated types (see Listing 1.6), therefore this term is slightly ambiguous as it may also refer to the actual annotation (for example `@rep`) of a type. Since this thesis is about the implementation of a Universe type system, annotations of types are typically used to represent the ownership modifier of a reference. Therefore *type annotation* refers to the type including type annotations (which is for example `List[AnyRef @any] @rep`) while *ownership modifier* means the actual annotation of a type (such as `@rep`).

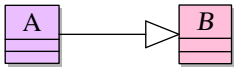
The term *main ownership modifier* or simply *main modifier* refers to the ownership modifier which is not part of the type arguments, e.g. `@rep` in `List[AnyRef @any] @rep`.

1.3.1 Notation in Diagrams

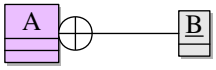
There are several diagrams contained in this thesis which use a UML-like notation to describe class hierarchies. The specific meaning of the elements of this notation is as follows:



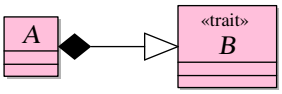
A class with three fields: `a` is of type `String` and public (indicated by the open lock), `b` is of type `Int` and protected (indicated by the semi-open lock), and `c` is of type `Boolean` and private (indicated by the closed lock). The method at the bottom is public, it takes an `Int` parameter and returns an `Int`.



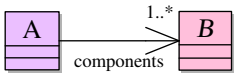
Class `A` extends the abstract class `B`: A specialization is expressed by an arrow with a triangular head and an abstract class' name is printed in italics. Fields and methods are not explicitly shown in this example, but they may exist – an even more condensed variant does not show the empty rectangles at all.



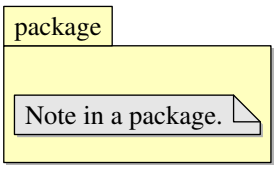
Class `A` contains an inner singleton object (or: *module*) `B`. If `B` was a class or trait, this would be an inner class or inner trait, respectively. Containment is expressed by the crossed circle on the side of the container. A singleton object's name is underlined. Such an object may also appear directly in a package, it does not need to be nested.



The abstract class `A` extends trait `B` whose self-type is `A`. As a trait is similar to an abstract class, its name is also printed in italics, but is has the additional `<<trait>>` stereotype. Trait `B` having the self-type `A` is expressed by the black diamond on the side of the actual self type.



Directed associations between classes are expressed by arrows. In this case class `A` has one or more references to instances of the abstract class `B`. The name of this association is `components`. A bidirectional association is expressed by a straight line. If the multiplicity is 1, it is not explicitly printed.



A note represents a comment in a diagram. It may appear everywhere, for example inside of a package like in this example.

1.4 Overview

The rest of this thesis is structured as follows. Chapter 2 gives an informal introduction to the Universe type system and subsequently explains the installation and usage of the implemented plugins. In Chapter 3, the architecture of the plugins gets introduced. This comprises an overview of the plugins' design and a detailed illustration of their class hierarchy. Chapter 4 discusses the actual implementation and Chapter 5 concludes. Appendix A provides some technical details which might be of use to someone who wants to modify or extend the implemented compiler plugins.

2 User's Guide for the Universe Type System Plugins

This guide serves as a quick reference to the Universe type system by providing an informal overview. It also explains the installation of the Universe type system plugins for Scala and their usage. The content of this guide is inspired by the quick-reference of the Universe type system implementation for JML [8].

2.1 Universe Type System

In Scala, every value is an object, it would therefore be possible to record an owner for every value. However, value types such as `Boolean`, `Int`, etc. are immutable. These types are therefore handled slightly different: In the static type checks, their default ownership modifier is `any` instead of `peer`, and at runtime, instances of immutable types do not get an owner assigned. They are not subject to the additional runtime checks, either.

2.1.1 Annotations

Ownership modifiers express object ownership relative to the current receiver object `this`. The implementation of ownership modifiers for Scala uses its support for annotations on types which allows the extension of Scala's normal type system.

2.1.2 Types

Figure 2.1 shows the ownership modifiers which are used in this implementation of the Universe type system. It also displays the relationship between the modifiers which could be seen as some kind of a subtype relation between ownership modifiers. Main modifiers can be used anywhere in the type while `some` is only allowed in type arguments. The internal modifiers cannot be written down in the program, they are only used internally by the type checker. They may nevertheless become visible in error messages and warnings from the plugins. The meaning of the modifiers is as follows:

`any` An `any` reference may point to arbitrary contexts. It is read-only when the owner-as-modifier property is being enforced.

`some` This modifier can only be used in type arguments. It is similar to `any`, but more restrictive except for subtyping, where `some` is more flexible.

`lost` The internal `lost` modifier indicates that ownership information got lost, for example during a viewpoint adaptation.

`rep` A `rep` reference expresses that an object is owned by `this`.

`peer` A `peer` reference expresses that the referenced object has the same owner as the `this` object.

`this` This internal modifier is only used as a main modifier for the current object `this` in order to distinguish accesses through `this` from other accesses.

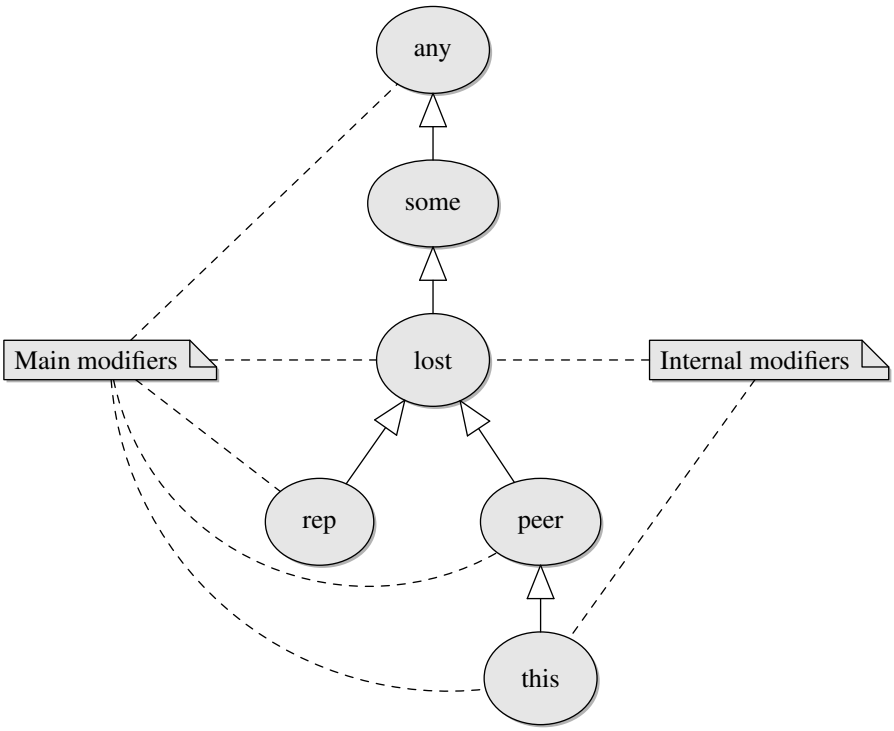


Figure 2.1: Subuniversing of ownership modifiers.

2.1.3 Methods

To indicate that a certain method does not have any side effects it can be marked as *pure*. Such methods do not modify any existing objects and may therefore be called on objects in arbitrary contexts, even when the optional owner-as-modifier property of the Universe type system is enforced.

2.1.4 Viewpoint Adaptation

The ownership expressed by ownership modifiers is always relative to `this`, hence it must be adapted if this viewpoint changes. This is for example the case for a field access like `x.f`: Here the types and especially the ownership modifiers of both `x` and `f` need to be taken into account when determining the owner, i.e. the type, of the referenced object:

- If `x` is `this`, then the ownership modifier of `x.f` (i.e. `this.f`) is the same as the one of `f` since it already is relative to `x`, which is `this` in this case.
- If the types of both references are `peer` types, this means that the object referenced by `x` has the same owner as `this`. As `f` is a `peer` reference of `x` and therefore has the same owner as `x`, the type of `x.f` must consequently be `peer`, too.
- If `x` has a `rep` type and `f` a `peer` type, then `x.f` yields a `rep` type. This is because `x` is owned by `this` and since `f` is of a `peer` type when seen from `x`, it has the same owner as `x`, which is `this`.
- If the type of `f` is an `any` type, the expression `x.f` also yields an `any` type since the owner of `f` is statically unknown.
- In all other cases, ownership information got lost, therefore the type of `x.f` is a `lost` type.

Method calls are handled by the same rules. This is especially relevant for Scala where every field access gets mapped to a call of a getter or setter method on the receiver object.

2.1.5 Subtyping

The subtype relation of the Universe type system extends the subtype relation of Scala by using the additional ownership modifiers. If two types are subtypes in Scala and have the same ownership modifiers, they are also subtypes in the Universe type system. Since there is also a subuniverse relationship between ownership modifiers as depicted in Figure 2.1, an additional subtype relation is possible: A `peer` or `rep` type, for example, is a subtype of the corresponding `any` type since it contains more specific ownership information.

2.1.6 Generics

Scala and the implementation of the Universe type system for Scala both support generics. The general ideas concerning viewpoint adaptation and subtyping are the same as mentioned above for non-generic types, but obviously more involved. See for example [6] for a detailed description of generics in the context of the Universe type system.

Scala also supports variance annotations of type parameters of generic classes. This is currently not supported in this implementation of the Universe type system.

2.1.7 Casts and Instanceof

If a read-write reference was passed out as a read-only reference, it may be required to cast it back down later in order to modify the referenced object. This cast operation is also supported by the implementation of the Universe type system. It is implemented by adding an additional check to calls of Scala's `asInstanceOf[T]` function. If the cast fails, a `ClassCastException` is thrown. Calls to `isInstanceOf[T]` are also modified in order to consider ownership.

2.1.8 Dynamic Checks

Typecasts and instanceof operations require additional checks and information about the owner of objects at runtime. Therefore some instructions for the runtime checks and the maintenance of the required information are added during the compilation of programs.

2.2 Installation of the Plugins

The following subsections assume that the environment variable `$SCALA_HOME` refers to the directory where the Scala installation resides. This variable is therefore synonymous to the `$JAVA_HOME` environment variable from Java. It is further assumed that the variable `$UTS_HOME` points to the directory which contains the Java archive files of the plugins and the accompanying libraries.

2.2.1 Requirements

The plugins were developed and tested with Scala 2.6.1. Later versions might work as well but as the plugins are highly dependent on the data structures used in the compiler, subtle breakage can occur. Some classes make use of Java's generics, therefore a version of Java supporting them is also needed, i.e. Java $\geq$ 1.5. The plugins were developed and tested with Java 1.5 and 1.6. When installing the source version of the plugins, Ant $\geq$ 1.6.3 is necessary for the build.

2.2.2 Binary

A binary distribution of the Universe type system plugins contains several Java archive files, namely the following:

`uts-static.jar` This file contains the plugin for the static Universe type system checks.

`uts-runtime.jar` This file contains the plugin which adds runtime checks to the compiled programs.

`uts-annotations.jar` This file provides the annotation classes. It is required when compiling or executing programs which were annotated with ownership modifiers.

`uts-rt-sc.jar` The support classes for the runtime checks, implemented in Scala. This file's classes must be available in the class path when the runtime checks are generated for the Scala implementation of the runtime support library.

`uts-rt-mj.jar` The support classes for the runtime checks, implemented in Java by [26], and part of MultiJava [32]. This file's classes must be available in the class path when the runtime checks are generated for the Java implementation of the runtime support library. Doing this results in interoperability with the Universe type system implementation for JML [7, 26].

`uts-scala-src.jar` The source code for the plugins, the annotations, and the runtime libraries.

Both plugin archives contain all classes required for the plugins themselves. The annotations must always be available in the class path because the input to the compiler may contain annotations and since the generated class files are decorated with even more annotations.

For the runtime support libraries this is similar. However, as it is possible to choose the library which should be used during the additional runtime checks with a compile-time option, it suffices if the library matching the chosen target library is available in the class path.

As these libraries must be available whenever a program compiled with the Universe type system plugins is to be executed, they must be shipped together with the program.

The installation of the above files is straightforward: The plugins ought to be copied to a directory where they are easy to find and access, for example `$SCALA_HOME/plugins/uts`. If the plugins should always be used automatically[1], they may also be copied to the directory `$SCALA_HOME/misc/scala-devel/plugins`.

Scala adds all libraries in `$SCALA_HOME/lib` to the class path, therefore copying the other Java archive files (except for the one with the source code) into this directory avoids the need to explicitly add them each time. The source code is not required in order to use the plugins.

2.2.3 Scala Bazaars

Scala Bazaars [28] is a package management system for Scala. By using the `sbaz` tool it is possible to install, update, and remove Scala extensions like libraries and plugins. If the URL of a Scala Bazaar providing a package for the Universe type system plugins is known, a universe descriptor like the one in Listing 2.1 can be stored in a file. Passing it to `sbaz setuniverse <file>` will then change the universe `sbaz` uses and update the list of available packages. The descriptor also contains the `scala-devel` universe which provides final releases of the Scala distribution and some third-party software. Now executing `sbaz available` should yield a list of the available packages which also contains the `uts-scala` meta-package. This package depends on all the other packages related to the Universe type system plugins, i.e. the plugins, the annotations, the runtime support libraries, the API documentation, and the source code.

In order to install the plugins, it suffices to execute `sbaz install uts-scala` which will download and install all the packages into the local managed directory. This is usually the directory `$SCALA_HOME`. Removal of a package is possible using the `sbaz remove <package-name>` command. Unfortunately, this can be a little tedious as one can only remove a package if no other package depends on it. The command `sbaz installed` prints the list of installed packages.

[1] This is not recommended, as they cannot cope with every Scala construct yet.

```
<overrideuniverse>
  <components>
    <simpleuniverse>
      <name>uts-scala</name>
      <location><!-- URL to the uts-scala Scala bazaar --></location>
    </simpleuniverse>
    <simpleuniverse>
      <name>scala-dev</name>
      <location>http://scala-webapps.epfl.ch/sbaz/scala-dev</location>
    </simpleuniverse>
  </components>
</overrideuniverse>
```

Listing 2.1: Bazaar descriptor for `scala-devel` and the Universe type system plugins.

2.2.4 Source

The build process of the plugins is based on Apache Ant. Since the build script depends on Scala, it checks if `$SCALA_HOME` is defined and if yes, this directory is used to look for the Scala compiler. In case this environment variable is not set, it will try to use the Scala compiler in `${user.home}/sbaz`.

After setting up the environment, it suffices to call `ant` in the top-level directory of the source tree, i.e. where the file `build.xml` resides. This builds the binary distribution mentioned in Section 2.2.2 except for the Java archive with the source code. The resulting files can be found in the `target` directory and may be installed accordingly.

The build script also supports a `run.tests` target which may be used to execute a set of test cases. Most of these tests compile a program using the plugins. The programs contain a known number of errors and warnings: A test is therefore successful if the number of detected errors matches the number of expected errors.

2.3 Usage

The following subsections explain how the plugins are actually used. This usually consists of three distinct actions: (1) At the beginning, a program gets manually decorated with ownership modifiers. In many cases, however, ownership modifiers do not need to be specified explicitly: They can usually either be inferred automatically or the default of `peer` is acceptable. The program can subsequently be (2) compiled and (3) run.

2.3.1 Annotations

As mentioned in Section 2.1.1 ownership modifiers are implemented by using Scala's annotations on types. The classes of the implementation reside in the `ch.ethz.inf.sct.uts.annotation` package. It is provided in the `uts-annotations.jar` file which therefore must be on the class path when using the annotations. For an example of the usage, see Listing 2.2: After importing the complete contents of the package, the ownership modifier annotations may be used by writing them on the right hand side of a type. This is a minor syntactical difference from other work (e.g. [7, 6]) that may require some adaption. Unfortunately, this can also make certain expressions hard to read. It also requires many brackets, as can be seen in the example. Some of them may be omitted, though, but it is better to always write them down.

One of the trickier examples where annotations are used can be seen in the definition of field `d` of class `C`: In this case, a new `Generic[Cls @any]` instance is created, and the reference to this instance is a `rep` reference. The constructor gets called with a `rep` reference to an instance of class `Cls`.

```
package utsdemo

// Import the annotations
import ch.ethz.inf.sct.uts.annotation._

// Application which creates a new instance of class C
object Main extends Application {
  new (C @rep)
}

// An example class which uses ownership modifiers
class C {
  // Use the annotations
  val a = new (Cls @rep)
  var b : Cls @any = new (Cls @rep)
  val c = new (Cls @peer)
  b = c

  // More complex example with annotations on the type arguments,
  // the type and the constructor argument.
  val d = new (Generic[Cls @any] @rep)(new (Cls @rep))
  b = d.field

  // Print a message to say that initialization is done
  println("Done.")
}

class Cls

class Generic[T <: Any @any](a: T) {
  val field = a
}
```

Listing 2.2: Example program which uses ownership modifier annotations.

In order to indicate to the plugins that a certain method may be considered as pure, a `@pure` annotation is also available. However, using this annotation does not result in any purity check of the annotated method. It is the programmer's responsibility to make sure there are no side effects.

2.3.2 scalac

To employ a plugin during the compilation with `scalac`, one can use the `-Xplugin:<path to plugin.jar>` option. This option may be used several times, i.e. once for each plugin which should be loaded. Alternatively one could also provide the option `-Xpluginsdir <path>` with a path to the directory containing the plugins. If no further options are given, `scalac` will print out a list of available command line arguments for the `scalac` compiler and the selected plugins.

In addition to the above options, it is necessary to pass `-Xplug-types` and `-Ygenerics` to `scalac` in order to make it process annotations on types and to enable support for Java's generics[2], respectively. If the `uts-annotations.jar` file was not copied to `$SCALA_HOME/lib`, it is also required to pass the path to this Java archive in the `-cp` option. The same applies for the `uts-rt-sc.jar` and `uts-rt-mj.jar` files, depending on the target library of the runtime checks. Putting it all together, this yields a command like

```
$ scalac -cp $UTS_HOME/uts-rt-sc.jar:$UTS_HOME/uts-annotations.jar \
    -Xpluginsdir $UTS_HOME -Xplug-types -Ygenerics <sourcefiles>
```

for the compilation using the Universe type system plugins.

2.3.3 Plugin Options

The plugins provide several options which can be used during compilation. Most of these options change the verbosity of the plugins, but some of them also affect their behavior. Options for a compiler plugin are specified by passing `-P:<plugin-name>:<option-value>` to the Scala compiler. There are two possible values for `<plugin-name>` when using the Universe type system plugins: `uts-static` for the plugin which checks the type rules statically and `uts-runtime` for the plugin which adds runtime checks.

Common Options

All those options related to verbosity are available in both plugins, as well as one option which affects the behavior.

`-P:<plugin-name>:[no]browser` Display a Swing-based browser which shows the abstract syntax tree after it has been processed by the plugin.

`-P:<plugin-name>:[no]ast` Print the abstract syntax tree after processing it in the plugin. This uses the `toString: String` method of the tree.

`-P:<plugin-name>:loglevel=<level>` The plugins contain a logging facility which supports different log levels. There are five levels, in ascending order of severity: `debug`, `info`, `notice`, `warn`, and `error`. Level `notice` is the default level. The last two levels will also print an excerpt of the source code which is related to the message. By specifying a certain log level, all messages below this level are suppressed. Hence, it is not possible to ignore errors.

`-P:<plugin-name>:defaults=<class>` This option allows to pass the name of a class which implements the `ch.ethz.inf.sct.uts.plugin.common.UTSDefaults` trait. These defaults will then be used during compilation. The option differs from the others in that it affects both plugins if it is specified for the `uts-static` plugin only. It is possible, though not recommended, to use different defaults for the two plugins.

[2] These options will not be present anymore in Scala > 2.6.1 where this is the default.

Options for the Static Type Checks

In addition to the compile-time selectable defaults, the plugin for the static checks also supports the `-P:uts-static:typerules=<typerules>` option. This allows to use a different implementation of the abstract `ch.ethz.inf.sct.uts.plugin.staticcheck.TypeRules` class, that is, a different variant of the type rules.

An actual implementation of the type rules may also provide additional options for the compiler. In the case of the default type-rules implementation, this is the `-P:uts-static:oam` option which enables the static checks of the owner-as-modifier property.

Options for the Addition of Runtime Checks

As already mentioned in Section 2.2.2, there are two different runtime support libraries. The `-P:uts-runtime:runtime=<library>` option therefore allows to choose the actual target library the runtime checks should be generated for. `<library>` can be either `sc` for the Scala implementation, which is the default, or `mj` for the MultiJava implementation. Both implementations are largely equivalent, although the Scala implementation lacks the additional implementations of [26] which for example allow a visualization of the object store.

2.3.4 `ant`

Listing 2.3 shows an example of an Ant build script which makes use of the compiler plugins. It employs both plugins in order to do the static checks and to add runtime checks to the compiled classes. In addition it uses a directory layout for the source tree and the target directory which was adopted from the standard directory layout of Apache Maven [33]. This should ease a future migration of a project to Maven using the Scala plugin [3] if it ever provides proper support for Scala's compiler plugins.

The build script consists of four parts: (1) First, several properties are set. The most important ones are `scala.home` and `uts.home` which point to the directories where Scala and the Universe type system plugins, respectively, can be found. These properties are set in such a way that they point to the directories stored in the `$SCALA_HOME` and `$UTS_HOME` environment variables, respectively, if they are set. If not, they get initialized to a directory below `${user.home}/sbaz`. Several other defaults are also provided using properties, which allows overriding on the command line using `-D<propertyname>=<value>`. (2) The class path contains the Scala compiler and library as well as the runtime support libraries and annotations of the Universe type system plugins. (3) In order to load the `<scalac/>` and other Scala related Ant tasks, the `<taskdef/>` declaration is used. It includes the file `antlib.xml` from the Java archive with the Scala compiler. (4) Finally, the `build` target compiles the source files using the compiler plugins.

When starting a new application which makes use of the Universe type system, this build script may serve as a starting point. It provides everything which is required to build the application and could e.g. be extended to execute unit tests.

2.3.5 Running the Compiled Application

After the program has been compiled successfully, it can be executed using `scala`. As mentioned before, the annotation classes and the runtime support library the runtime checks were built for must be on the class path. This yields a command like

```
$ scala -cp $UTS_HOME/uts-rt-sc.jar:$UTS_HOME/uts-annotations.jar:.  \
    <main-class>
```

to execute the main class of a program.

Of course it is also possible to execute the compiled program with Ant. This is especially useful if Scala's SUnit is used for the implementation of unit tests as these tests are encapsulated in Scala applications. Listing 2.4 shows an Ant build script which includes the script from Listing 2.3. It provides an additional `run` target and a preset definition which declares a new `<scala/>` task. This task can be used to execute the main class of a Scala program.

```
<?xml version="1.0" encoding="UTF-8"?>

<project name="Ant Example" default="build">
  <property environment="env" />
  <!-- Set scala.home to local sbaz-managed directory or $SCALA_HOME if set -->
  <condition property="scala.home"
             value="${env.SCALA_HOME}"
             else="${user.home}/sbaz">
    <isset property="env.SCALA_HOME" />
  </condition>

  <!-- Use plugins from ${scala.home}/plugins/uts or $UTS_HOME if set -->
  <condition property="uts.home"
             value="${env.UTS_HOME}"
             else="${scala.home}/plugins/uts">
    <isset property="env.UTS_HOME" />
  </condition>

  <!-- Input and output directories -->
  <property name="src.main" value="src/main/scala" />
  <property name="target" value="target" />
  <property name="target.classes" value="${target}/classes" />

  <!-- Options for the compiler -->
  <property name="options.plugins"
            value="-Xplugin:${uts.home}/uts-static.jar -Xplugin:${uts.home}/
                uts-runtime.jar" />
  <property name="options.plugin"
            value="-P:uts-static:loglevel=info -P:uts-runtime:loglevel=info" />
  <property name="options.scalac" value="-Xplug-types -Ygenerics" />

  <!-- Construct build.classpath for use during compilation -->
  <path id="build.classpath">
    <pathelement location="${scala.home}/lib/scala-library.jar" />
    <pathelement location="${scala.home}/lib/scala-compiler.jar" />
    <!-- Library with runtime support classes -->
    <pathelement location="${uts.home}/uts-rt-sc.jar" />
    <pathelement location="${uts.home}/uts-rt-mj.jar" />
    <!-- The annotations for the ownership modifiers -->
    <pathelement location="${uts.home}/uts-annotations.jar" />
  </path>

  <!-- Include ant-support from Scala -->
  <taskdef resource="scala/tools/ant/antlib.xml"
           classpathref="build.classpath" />

  <!-- Compile the example program -->
  <target name="build" description="Compile the source.">
    <echo level="info">Scala location:  ${scala.home}.</echo>
    <echo level="info">Plugin location: ${uts.home}.</echo>
    <mkdir dir="${target.classes}" />
    <scalac srcdir="${src.main}"
            destdir="${target.classes}"
            classpathref="build.classpath"
            addparams="${options.scalac} ${options.plugins} ${options.plugin}"
            includes="**/*.scala">
    </scalac>
  </target>

  <target name="clean" description="Remove the target directory.">
    <delete dir="${target}"
            quiet="yes" />
  </target>
</project>
```

Listing 2.3: Example `build.xml` for use with Scala and the Universe type system plugins.

```
<?xml version="1.0" encoding="UTF-8"?>

<project name="Ant example to run a Scala program" default="run">
  <import file="build.xml"/>

  <!-- Declare some presets for the predefined java task, accessible via
      the new scala task -->
  <presetdef name="scala">
    <java fork="true" failonerror="true" classpathref="build.classpath">
      <arg line="${options.scalac}" />
      <classpath>
        <pathelement location="${target.classes}"/>
      </classpath>
    </java>
  </presetdef>

  <!-- Run the main class -->
  <target name="run" depends="build">
    <scala classname="utsdemo.Main" />
  </target>
</project>
```

Listing 2.4: Example `run.xml` for use with the `build.xml` from Listing 2.3. These files can also be merged.

3 Architecture

This chapter presents the architecture of the Universe type system implementation for Scala. It starts with a general overview and subsequently shows how ownership modifiers are represented. It finally discusses the design of the two plugins which together implement the static type checks as well as the runtime checks.

3.1 Overview

The Universe type system covers static type safety as well as a runtime model. It is therefore sensible to mirror this separation in an implementation of the type system. Hence, there are two plugins for the Scala compiler: The first one checks the type rules statically and the second one adds runtime checks to the program.

As Scala usually infers the type for variables, type arguments, and method return types, the first plugin also infers ownership modifiers. A condensed overview of the architecture may be found in Figure 3.1.

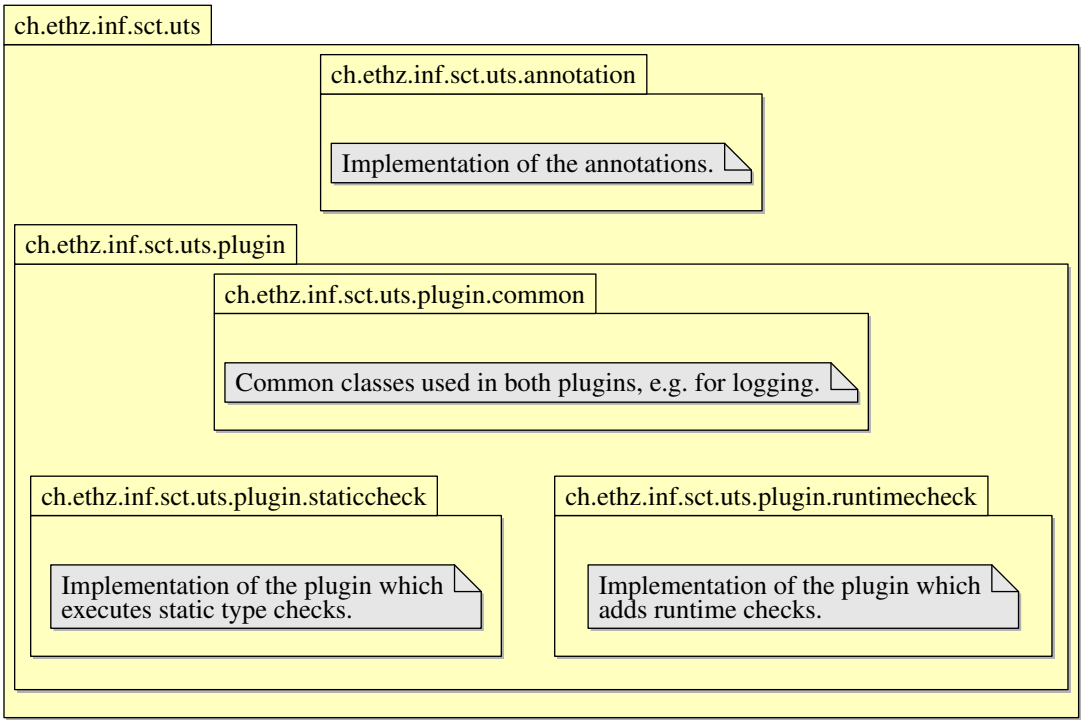


Figure 3.1: Overview of the architecture.

The implementation of the type system consists of two major parts which are distributed over two packages:

1. Ownership modifiers such as `any`, `peer`, and `rep` are represented by annotations. The classes implementing these annotations are collected in the `ch.ethz.inf.sct.uts.annotation` package which is therefore the only package a user of this Universe type system implementation usually notices.

2. The actual implementation of the plugins consists of three prominent packages:

   a) A common package with classes used in both plugins. This comprises a class for logging and a class with defaults used during the execution of the plugins. A user of the plugins may provide an alternate implementation of these defaults which are then used during compilation. This allows a modification of certain aspects of the compiler's behavior. It comprises for example the default ownership modifiers which should be used if no modifier has been specified explicitly in `new` expressions or method parameters.

   b) The plugin for the static type checks. It allows the user to select the actual implementation of the type rules which should be employed during compilation. This way it is possible to use different variants of the Universe type system by specifying a compile-time option.

   c) The plugin for the addition of the runtime checks. It modifies the program in order to add additional checks which will be executed at runtime. Since these checks require information about the ownership relation, it also inserts code which maintains this topology.

Figure 3.2 shows where these two plugins fit into the structure of compiler plugins introduced in Section 1.1.6. The abstract base classes for the Universe type system plugins provide some common functionality: The abstract `UniverseTypeSystemPlugin` handles command line options and usually passes them to implementations of the abstract `UniverseTypeSystemComponentBase` class. This abstract class provides a logging facility, several helper functions, and also stores the values of certain command line options.

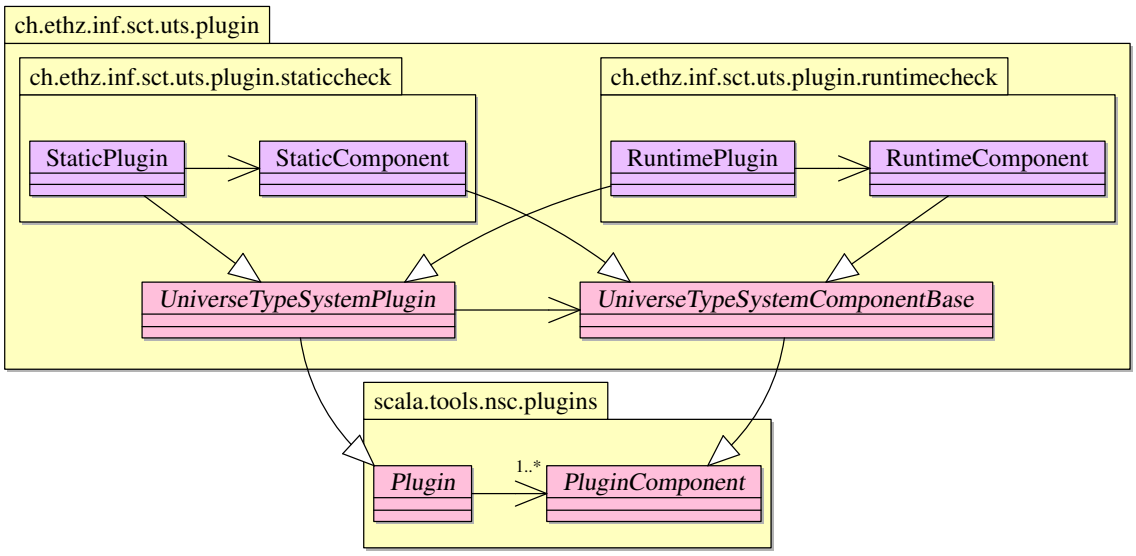


Figure 3.2: Hierarchy of the plugins in the context of the general layout for Scala's compiler plugins from Figure 1.2.

3.2 Ownership Modifiers

Ownership modifiers such as `any`, `peer`, and `rep` are represented using source code annotations in the program. Some modifiers are only used internally, e.g. `lost`. This difference can be modeled by using the hierarchy in Figure 3.3: The `OwnershipModifier` trait lies at the root.

`OwnershipModifierAnnotation` extends `scala.StaticAnnotation` and its specializations may therefore be used in the program's source code. Static annotations are preserved in the final class file such that they are available to the type checker across different compilation units.

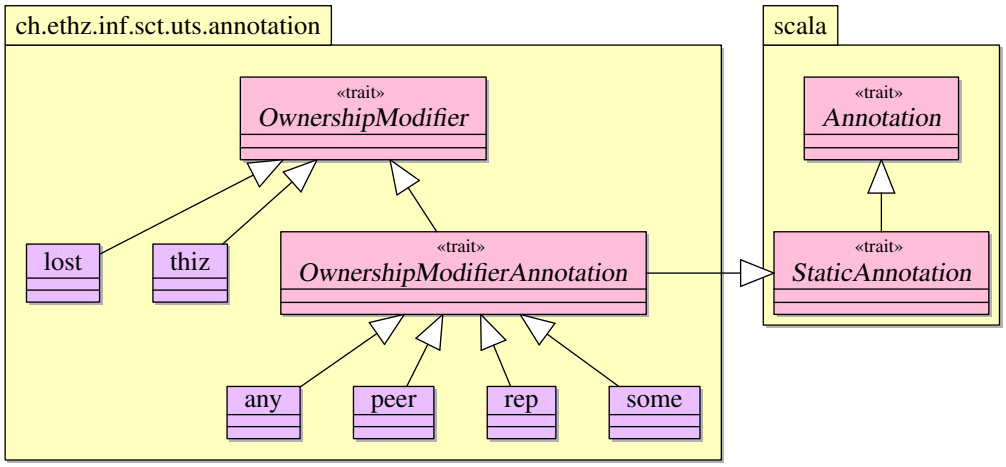


Figure 3.3: Class hierarchy for ownership modifiers.

As mentioned in Section 3.1, the implementation supports different variants of the Universe type system. The meaning and behavior (for example in viewpoint adaptations) of ownership modifiers usually depends on this variant. Therefore most of the operations an `OwnershipModifier` provides are declared in an `OwnershipModifierExtender` trait's abstract `ExtendedOwnershipModifier` class. Since the trait is mixed into the `TypeAbstraction` trait which finally gets mixed into the abstract `TypeRules` class, the `ExtendedOwnershipModifier` is available when implementing actual type rules. Figure 3.4 shows this hierarchy and also includes the concrete default implementation. This diagram is an excerpt of Figure 3.5 and only shows the classes relevant for the handling of ownership modifiers.

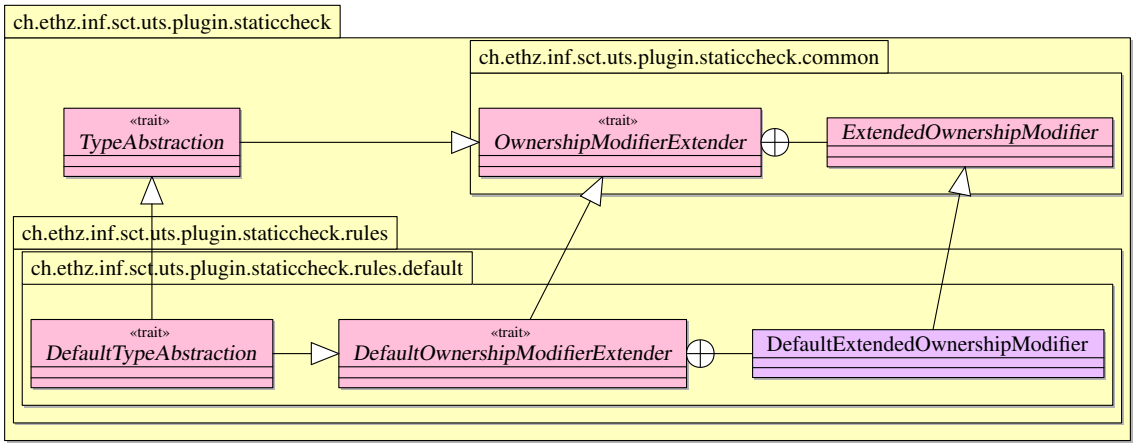


Figure 3.4: Simplified class hierarchy for extended ownership modifiers, extracted from Figure 3.5.

The `OwnershipModifierExtender` trait also provides an implicit and abstract conversion function from an `OwnershipModifier` to an `ExtendedOwnershipModifier`. This allows actual implementations of the `OwnershipModifierExtender` trait to implicitly create instances of a concrete implementation of the abstract `ExtendedOwnershipModifier` class with a specialized behavior.

In the default implementation of the Universe type system's rules from Figure 3.4, this specialized behavior is implemented by the `DefaultExtendedOwnershipModifier` class. It provides for

example the actual viewpoint adaptation function for ownership modifiers.

In order to get access to the class with the concrete implementation of the behavior of ownership modifiers, the `DefaultOwnershipModifierExtender` trait gets mixed into the `DefaultTypeAbstraction` class. This so-called *mixin-class composition* also results in the implicit conversion function being available via the `DefaultTypeAbstraction` trait. Concrete implementations of the abstract `TypeRules` class mix in their corresponding `TypeAbstraction` trait. It is therefore possible to seamlessly use an `OwnershipModifier` in the type rules as if it was an `ExtendedOwnershipModifier` since this conversion takes place implicitly.

3.3 Static Checks

Figure 3.5 shows the class hierarchy of the plugin for the static type checks. This plugin is designed in a way which allows the implementation of different variants of actual type rules. It provides a default implementation of these rules in its `ch.ethz.inf.sct.uts.plugin.staticcheck.rules.default` package. The rule set which should be used during the type checks may be chosen at compile time by specifying a concrete implementation of the abstract `TypeRules` class.

The diagram shows that the plugin consists of several traits, most of which declare one or more inner classes. This architecture is similar to the Scala compiler [22] and it is also partially necessitated by the compiler's design: As mentioned in Section 1.1.6, the `global` reference provides access to the types used for Scala's type representation and other types used in the compiler. Since these types are only accessible through the `global` reference, it is for example not possible to implement a top-level class with a parameter of such a type. This issue can be solved by specifying an abstract member `global` of type `scala.tools.nsc.Global` in the `UniverseTypeRepresentationBase` and `TypeAbstraction` traits. The member is then implemented in the abstract `TypeRules` class which gets the `global` reference from the `StaticComponent`.

3.3.1 Type Representation

The representation of types in the Scala compiler is more complex than the minimal calculi usually found in papers and Section 1.2.1. The `TypeAbstraction` trait therefore supplies a suitable abstraction from the internal representation of Scala. This representation reflects the notation from Section 1.2.1: A type is either a non-variable type or a type variable identifier with an upper bound. Non-variable types consist of a main ownership modifier, the name of the usual Scala type, and a possibly empty sequence of type arguments.

At the base of the abstraction lies the abstract `UType` class which specifies a common interface for concrete implementations. It includes operations for the viewpoint adaptation and a subtype test as well as a function to test the type for well-formedness. There are two classes to represent the two different kinds of types and some classes used for error handling.

Non-variable types These types are represented by the abstract `NType` class. In addition to the common `UType` methods, it also contains functions for the handling of ownership modifiers (i.e. to substitute or query them) and type arguments (i.e. to substitute or instantiate them).

Type variable identifiers The abstract `TVarID` class represents type variable identifiers and includes a function to get the upper bound of the type variable.

Type errors Whenever an operation which should return a `UType` detects an error, it returns an instance of the abstract `ErroneousType` class containing a textual reason for the error. This may for example happen during the test for well-formedness of a type. The general idea behind this concept is to avoid the use of exceptions. It is inspired by the `scala.Option` class which represents optional values and may also be used for error handling.

There are two concrete implementations of the `ErroneousType`: `InvalidType` is used for a single error and contains therefore only one reason, whereas an `InvalidTypes` instance may

contain several `InvalidType` instances. This is used in methods checking rules where more than one error may be found, for example in the type parameters of a method definition.

The `StaticErrors` trait is also used during error handling. It provides some functions to create standardized error messages for inclusion in an `InvalidType`'s textual reason. This is used for example in the case of type errors where a different type was found than expected.

3.3.2 Method Representation

Section 1.2.1 also uses a general notation for method signatures. It comprises a purity indicator, type and method parameters, and the return type. In order to mirror this abstraction, the `UTSMethodSignature` provides the abstract `MethodSignature` class. It offers methods to resolve type variables, to substitute type parameters and to compare two method signatures for equality.

3.3.3 Assembling the Abstraction of the Compiler's Type Representation

When looking at the class diagram in Figure 3.5 one can see that the above abstraction is spread over several traits, abstract classes, and also some concrete classes. Among these traits, the `TypeAbstraction` trait is particularly central: It mixes in all traits which are used to abstract from the Scala compiler's type representation and the `OwnershipModifierExtender` explained in Section 3.2. In addition, it is also declared as being the self type (see Section 1.1.5) of all those traits which are part of the abstract type representation. It may therefore be compared with the abstract `SymbolTable` class of the Scala compiler.

Using abstract type members, mixin class composition, and self types can generally result in a modular architecture based on reusable components [22]. The architecture of the plugin for the static type checks tries to utilize this in order to be extensible with different variants of the Universe type system. A concrete implementation of type rules would therefore specialize the traits which contain abstract classes. It may then provide concrete implementations of these abstract classes which are usable in the type rules. This can be seen at the bottom of Figure 3.5: The `DefaultTypeAbstraction` trait mixes in the `TypeAbstraction` and those traits which provide implementations of the abstract classes. By extending the `DefaultTypeAbstraction`, the concrete implementation of the type rules in `DefaultTypeRules` gains access to these implementations.

The `DefaultTypeRules` class could therefore be compared to the `scala.tools.nsc.Global` class which extends the aforementioned `SymbolTable`: Both are central in the particular applications and they both mix in a type representation.

Distributing the implementation into several traits and classes and subsequently mixing them together also makes it possible to spread the source code over multiple files. This is also done in the Scala compiler itself.

3.3.4 Checking Type Rules

Actual checks of the type rules are done in concrete implementations of the `TypeRules` trait. This trait specifies a checker function for each relevant language construct and comprises for example a `checkAssignment` and `checkMethodDefinition` function. These functions are called by the `StaticComponent` class during a traversal of the abstract syntax tree of the input program. The checker functions take arguments which allow them to verify if the type and well-formedness rules of the construct hold: This usually comprises the involved types, possibly a relevant compiler symbol, and sometimes also the position of a construct (e.g. the position of the *then* and *else* branches of the if-then-else expression).

All checker methods return a `UType` instance or a list thereof as result. Usually, this is the type of the checked expression or definition, but with adapted viewpoint. This type is then used in the `StaticComponent` for type inference and propagation. As mentioned in Section 3.3.1, `UType` instances are also used in error handling, so the return value may also express the violation of a rule from the type system. The `SymbolStates` trait which gets mixed into the `StaticComponent` is used in the type inference; see Section 4.2.2.

3.4 Runtime Checks

Runtime checks are required to make operations like type casts and instanceof checks Universe-aware. In order to achieve this, the ownership hierarchy must be built and maintained at runtime. It is then possible to use this information to conduct additional checks. Earlier work [26] already implemented this for Java in the MultiJava compiler [32].

The required modifications and extensions of the abstract syntax tree are performed in a second plugin. It adds the runtime checks and is smaller than the first plugin since it only needs to extend the abstract syntax tree in three situations:

Constructor methods Since newly created objects should be assigned an owner, additional instructions are inserted into the constructor methods to store the owner of the new object.

Object creation The object which initiated a method call is usually not known to the executed method. However, a constructor method must know the object which is creating the new instance, since it has to set the owner of the newly created object. It is therefore necessary that a reference to `this` gets stored whenever a `new` statement is encountered in the abstract syntax tree. The reference must be saved such that it is available in the constructor.

Typecasts and typetests Unlike Java, Scala does not support special language constructs for typecasts and typetests. Instead, these operations are provided via the two parameterized methods `asInstanceOf[T]: T` and `isInstanceOf[T]: Boolean` of class `scala.Any`. This class is at the root of Scala's class hierarchy and therefore these methods are available on every object.

As the runtime checks need to extend these operations to do additional examinations, calls to the respective methods must be modified as well.

Maintaining the information about owners and implementing the checks by directly modifying the abstract syntax tree alone would be neither simple nor feasible. Hence, additional runtime support libraries are used which are implemented in a high-level programming language. There are two such libraries, one of which is directly taken from earlier work in [26], while the other one is based on this same work. Both libraries are used for the management of ownership information and the additional checks at runtime. The first one is written in Java and it is part of the MultiJava compiler [32] while the latter is an adapted version of it, ported to Scala. They consequently both use a hash table to store the ownership relation. Two reasons led to the decision to support and use two different runtime support libraries:

Interoperability The implementation of the Universe type system for JML [7] uses the runtime support library from [26]. Generating code which uses this library results in interoperability with Java code compiled using the Universe type system implementation for JML. The methods which create calls to the methods of the Java version of the library are implemented in the `MJRuntimeCheckTransform` class.

Extensibility The class `RuntimeCheckTransform` extends the abstract syntax tree such that the methods of the Scala implementation are called at runtime. This allows extensions of the runtime checks by using features available to Scala only, without the need to modify the MultiJava compiler and possibly JML as well.

The target library for the runtime support can be selected at compile time. It defaults to the Scala implementation.

Figure 3.6 shows the class diagram of the architecture of this second plugin. Its implementation specializes the transformer provided by the Scala compiler in order to add runtime checks to the abstract syntax tree. A concrete implementation of the abstract `RTCTransformer` is contained in the implementation of the `RuntimeCheckTransformBase` trait for the respective library: The `RuntimeCheckTransformer` classes therefore provide specialized methods to create subtrees representing runtime checks which make use of the corresponding runtime library.

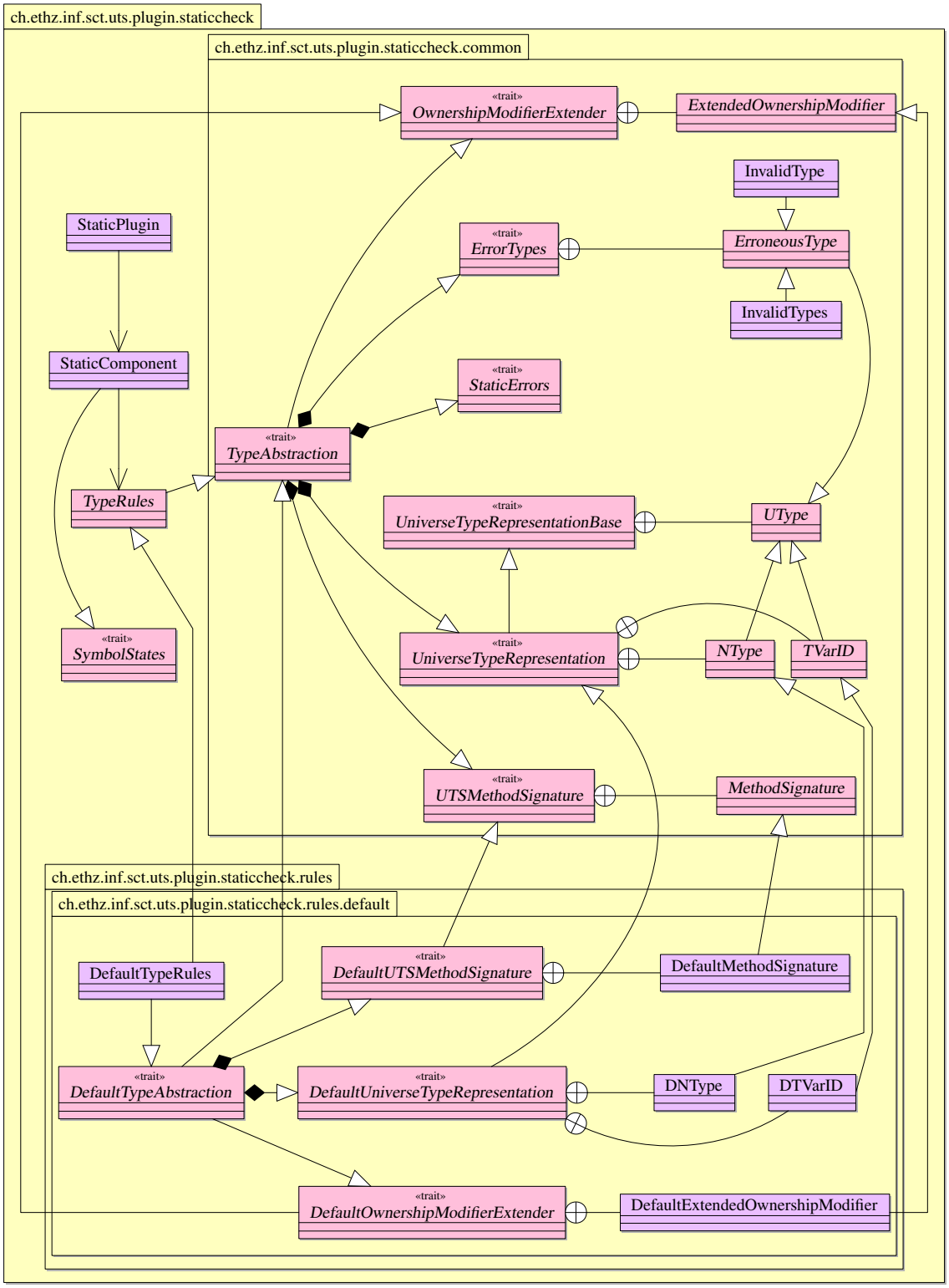


Figure 3.5: Plugin for the static checks.

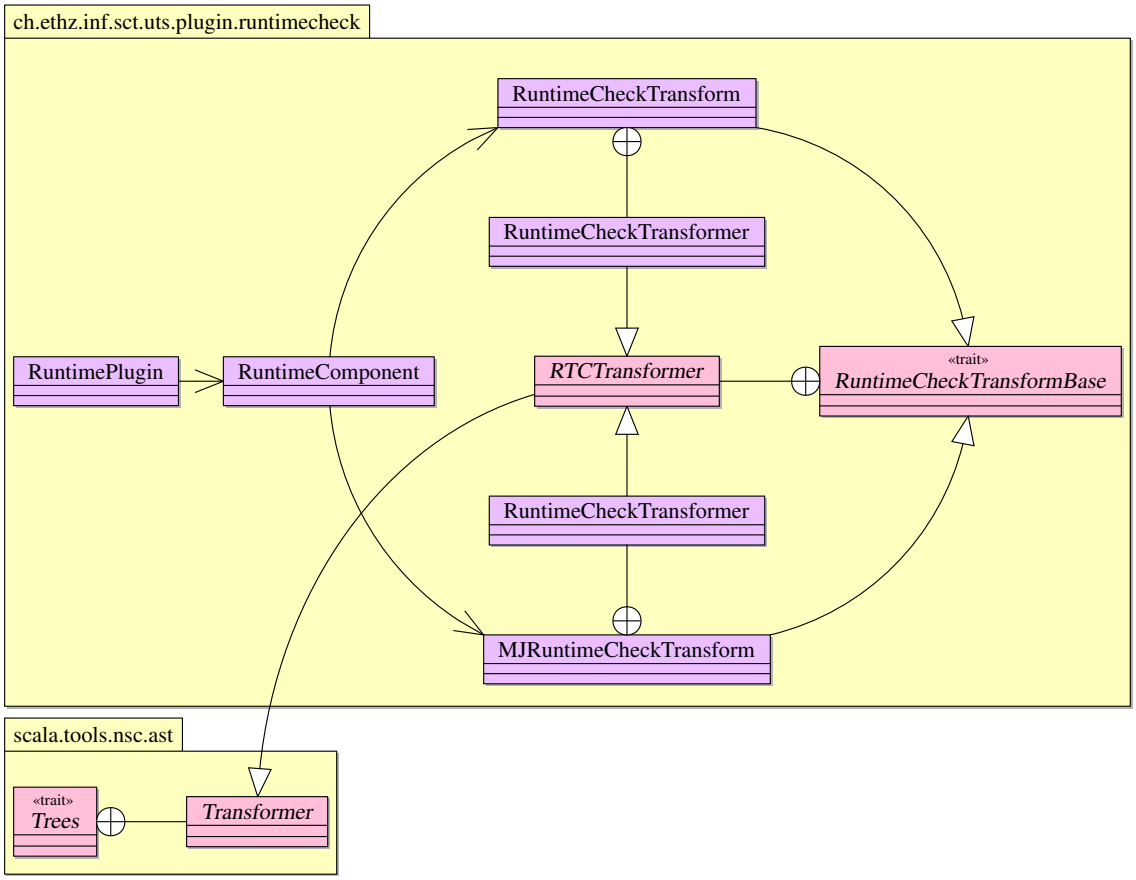


Figure 3.6: Plugin for the addition of runtime checks.

4 Implementation

This chapter presents the implementation of the plugins. It starts with an introduction to the abstract syntax trees used in the Scala compiler and shows the inference of ownership modifiers. It subsequently addresses the actual plugins and also explains the implemented logging facility and the test-suite.

4.1 Abstract Syntax Tree

The Scala compiler uses an abstract syntax tree (AST) to represent and process a program at compile time. The AST can be seen as an intermediate language. In the case of the Scala compiler, this language is a typed and desugared version of Scala itself [1].

A unit of compilation typically corresponds to one file of source code. Therefore, if more than one file is being compiled, there are also several compilation units, each of which contains a reference to the AST of the underlying file in its `body` field. All compilation units are gathered in a so-called *compilation run*.

The compilation process consists of a succession of code rewriting phases each of which further simplifies the code by replacing complex constructs by simpler ones. In the end this AST is converted to Java byte code and written to one or more class files.

Listing 4.1 shows a very simple Scala class which contains the definition of a final field and a method call with this field as a parameter. When the Scala compiler processes this file it generates an AST which resembles the simplified one in Figure 4.1.

4.1.1 Nodes of the Abstract Syntax Tree

Every node in the tree is an instance of the abstract class `scala.tools.nsc.ast.Trees.Tree`. All classes which implement this abstract class are case classes; it is therefore possible to do a pattern match over a tree. When working with trees, it is usually necessary to traverse the trees in some way. One could do it by hand, which is flexible but tedious. The problem is that one needs to consider every kind of node in the tree in order to do a complete traversal even if certain kinds are of no interest. Fortunately, this is typically not necessary as the Scala compiler provides two classes which may be extended and used: `scala.tools.nsc.ast.Trees.Transformer` and `scala.tools.nsc.ast.Trees.Traverser`. The former is used to match and replace complete subtrees while the latter may be used to perform side-effects on the nodes or to gather information. Here, side-effect means for example that the node's symbol may be updated or that its type could be changed. But it is not possible to remove, replace or add nodes using a traverser. By using these classes one only has to consider those nodes which are relevant to the application.

Listing 4.2 shows the `SimpleTraverser` class which is an example of a traverser. It is enclosed by an implementation of the Scala compiler's abstract `PluginComponent` class (see Section 1.1.6) and could therefore be part of a compiler plugin. The `traverse(tree: Tree): Unit` method traverses

```
class C {
  val str = "s"
  println(str)
}
```

Listing 4.1: Code to the AST from Figure 4.1.

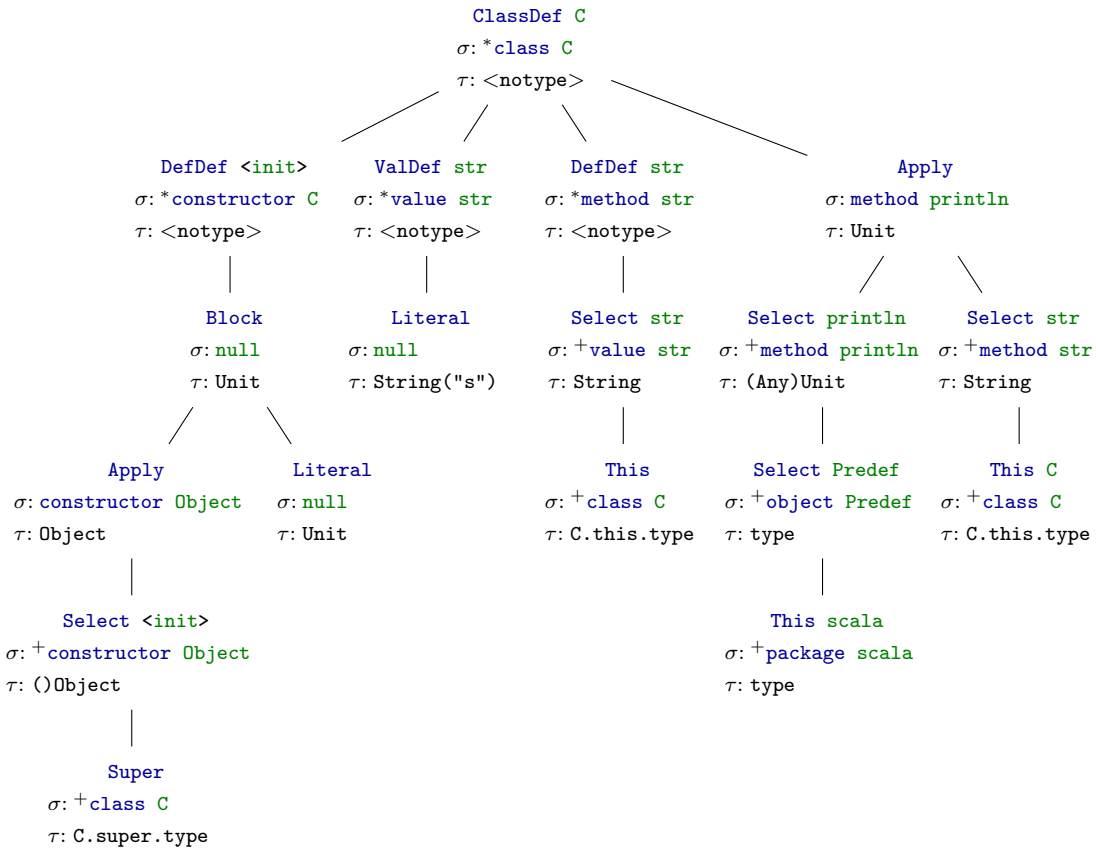


Figure 4.1: Simplified AST for the code in Listing 4.1 after phase `superaccessors`.

the AST and prints the names of methods and classes when it starts processing a corresponding definition. Processing of an AST starts when the `SimpleTraverser`'s `traverse` method gets called with the `body` reference of a compilation unit.

A transformer is similar, but it uses a `transform(tree: Tree): Tree` method for the traversal. The main difference is that the `transform` method must return a `Tree` instead of `Unit`: This makes it possible to modify the currently processed node by, for example, replacing its subtrees or by completely exchanging the node itself. The plugin which adds runtime checks (see Section 4.4) employs this in order to add new nodes to the AST.

4.1.2 Symbols and Types

Most nodes of the AST in Figure 4.1 refer to a compiler symbol σ which they either define (marked by *) or use in some way (unmarked or highlighted by $^+$). A symbol stores information about various source code constructs like fields, variables, methods, or classes. The information comprises an identifier for the symbol, its type, its kind (i.e. if it is a method, class, field, etc.), and its owner (i.e. the enclosing class, method, etc.).

Since the AST is typed, most nodes also have a type τ assigned. There is no distinction between statements and expressions in Scala, hence type τ is the type of the expression represented by the node. As definitions (represented e.g. by `ValDef` and `DefDef` nodes) are neither statements nor expressions, they do not have a type.

A simple symbol table for the program in Listing 4.1 is shown in Table 4.1. It displays the kind and the type of those symbols which were defined in the program. The type `()C` of `constructor C` means that the method does not take any parameters but that it returns an instance of class `C`.

```
package ch.ethz.inf.sct.uts.plugin.simpletraverser
import scala.tools.nsc._
import scala.tools.nsc.plugins._

/** Plugin component which runs after runsAfter. */
class SimpleTraverserPluginComponent (val global: Global)
  extends PluginComponent {
  import global._
  /** Name of this compiler phase. */
  val phaseName = "simpletraverser-phase"

  /** When to execute this phase. */
  val runsAfter = "refchecks"

  /** Factory to create the new phase for the graph generation. */
  def newPhase(prev: Phase) = new Phase(prev) {
    def name = phaseName

    /** Process the compilation units. */
    def run {
      currentRun.units foreach { unit =>
        (new SimpleTraverser).traverse(unit.body)
      }
    }
  }

  /** A simple traverser which prints the name of the currently processed
      class/method */
  class SimpleTraverser extends Traverser {
    /** Method containing the pattern match. */
    override def traverse(tree: Tree) {
      tree match {
        case ClassDef(mods, name, tparams, impl) =>
          println("Processing class definition of "+tree.symbol+"...")
        case DefDef(mods, name, tparams, vparamss, tpt, rhs) =>
          println("Processing method definition of "+tree.symbol+"...")
        case _ => ()
      }
      super.traverse(tree)
    }
  }
}
```

Listing 4.2: Example of a traverser.

The type of `method str` could also be written as `()java.lang.String` which is largely the same as `=> java.lang.String` in the case of this example: In the compiler, the type of `method str` is a so-called `PolyType`, but as it does not have any type parameters, it represents a parameterless method type. It is therefore basically the same as `()java.lang.String` which is a `MethodType` that is also parameterless in this case.

Symbol σ	Kind	Type of σ
`class C`	class	`C`
`constructor C`	method	`()C`
`value str`	variable	`java.lang.String`
`method str`	method	`=> java.lang.String`

Table 4.1: Simplified symbol table for the symbols defined in Listing 4.1.

When comparing the AST to the source code of the program, several differences and some interesting details emerge. The AST contains two method definitions which are not visible in the source code. One of them is the constructor method `<init>` which calls the super constructor from class `java.lang.Object`. In Scala there is usually no explicit constructor method in the program, instead the class body represents the default constructor. This is also the reason why a class may have arguments. The other one is method `str` which is the generated getter method for the field `str`. If `str` was not a final reference (i.e. a `var` instead of a `val` field), there would also be a method `str_=(x: String)` serving as a setter method for the field. As one can see in the `Apply` on the right-hand side, every access to a field is done through a call of the corresponding getter method.

Another notable detail about the AST in Figure 4.1 is that although there is an actual constructor method now, the initialization of `value str` and the call to the `println` method are not part of this method. Instead, they are directly under the node `ClassDef C`. The reason for this is that the compilation consists of a succession of code rewriting phases as mentioned in Section 4.1.

A snapshot of the AST after a given phase can be examined by using the `-Ybrowse:<phase>` option of the Scala compiler. In the case of this example, the snapshot was retrieved after phase `superaccessors`. This phase is one of the earlier phases during compilation and the phase after which the implemented plugins are executed. If the snapshot had been taken after phase `constructors`, the aforementioned subtrees would actually be below the `DefDef <init>` node with the constructor definition.

4.2 Inference of Ownership Modifiers

As mentioned in Section 1.1.3, the Scala compiler supports local type inference. But the Scala compiler knows nothing about ownership modifiers, the Universe type system or other, possibly more restrictive type systems at all. In order to support the Universe type system, ownership modifiers, implemented using Scala's annotations on types, must be handled, inferred if possible, and propagated through the abstract syntax tree. The inference of ownership modifiers is not strictly required to support the Universe type system. It nevertheless reduces the annotation overhead for the programmer and results in a smooth integration of the Universe type system with the normal Scala type system.

The algorithm for the inference of ownership modifiers should infer them for variables, method return types, and type arguments, i.e. it should infer them at the same places where Scala may already have inferred a type. But this is not enough: It also needs to take care of propagating all ownership modifiers throughout the abstract syntax tree since they are required when checking the Universe type system's rules. While doing so, it has to add them to the symbol types inferred by Scala such that they are stored persistently in the class file for later access.

Note that the inference implemented in this thesis is different from earlier work on Universe type inference in e.g. [14, 12, 2]. The implemented algorithm works only locally, that is, it infers the Universe type from the initialization. This is similar to what the Scala compiler does for

```
package ch.ethz.inf.sct.uts.examples
import ch.ethz.inf.sct.uts.annotation._
class A {
  val b /* : B @rep     */ = new (B @rep)
  val s /* : Cls @any   */ = b.s
  def t /* : Cls @peer  */ = new (Cls @peer)
  val u /* : Cls @rep   */ = b.t
}
```

Listing 4.3: Class A, stored in `A.scala`.

normal types (see Section 1.1.3). The inference does not try to create a good ownership structure either, it only takes the information which was provided by the programmer and propagates it such that the program can be compiled. Inferred ownership modifiers are used during compilation and written to the class files, but they are not added to the source code of programs compiled using these plugins.

4.2.1 Recursive Definitions and Ownership Modifier Propagation

There are two main difficulties which need to be resolved in order to successfully infer ownership modifiers:

Recursive definitions Recursively defined functions and fields in different classes which depend on each other may lead to cycles during the type inference. Scala already detects such cycles for non-annotated types and forces the programmer to specify the type explicitly in these cases. This unfortunately is of little help when inferring annotations for an extended type system like the Universe type system. Therefore an inference algorithm for ownership modifiers has to detect cycles itself and must force the programmer to provide annotations or use some default whenever it detects a cycle.

Propagation throughout the AST After the types have been inferred, they must be propagated through the abstract syntax tree to be usable during the type checks. Since the abstract syntax tree in the Scala compiler is a typed one, care must be taken to modify the compiler's type representation correctly. If any modification yields an abstract syntax tree which is not correctly typed anymore, later phases of the compiler may violate explicit or implicit assertions. Both cases result in internal compiler errors and a termination of the compilation.

To solve these problems, the implementation of the inference traverses the abstract syntax tree two times: The first time it extracts the symbols of value and method definitions which are stored together with some state information. This is followed by the second traversal which infers types and propagates them. At the same time, it also enforces the type rules.

The following explanations will use classes `A` and `B` from Listing 4.3 and Listing 4.4, respectively, as an example. These two classes depend on each other, it is therefore not possible to completely process one class before the other: Field `s` of class `A` depends on the type of field `s` from class `B` while the reverse applies for field `t` of class `B`.

Both classes already contain the final types in comments. The goal of the ownership modifier inference is to add the correct modifiers to unannotated types. These unannotated types were already inferred by the Scala compiler. Hence, inference of ownership modifiers should add them such that the types finally match those from the comments.

4.2.2 Symbol Extraction

While traversing the abstract syntax tree for the first time, a mapping from symbols to state information like the one in Table 4.2 is created. This mapping and the state information are saved

```
package ch.ethz.inf.sct.uts.examples
import ch.ethz.inf.sct.uts.annotation._
class B {
  val a /* : A @peer   */ = new (A @peer)
  val s /* : Cls @rep  */ = new (Cls @rep)
  val t /* : Cls @peer */ = a.t
  val u /* : Cls @any  */ = a.s
}
```

Listing 4.4: Class B, stored in `B.scala`.

in a map provided by the `SymbolStates` trait which is mixed into the `StaticComponent` class. The state stores the following information about a symbol:

Typed A symbol has been typed if the type of the symbol is known and if its definition was type checked. Therefore this is false in the beginning despite the fact that the type annotation without ownership modifiers was already inferred by Scala. It only refers to the inference of the ownership modifier and the type check of the Universe type system's rules. As soon as the definition of the symbol has been processed (i.e. the subtree referenced by the "Code" column), this is set to true.

Locked A symbol is locked if its type is currently being inferred, i.e. the referenced subtree is being processed. This is used to detect cycles during the inference: Whenever the type of a symbol is required and the symbol is not typed yet but locked, this means that there is a recursive definition. In this case the inference algorithm checks if the symbol's type annotation contained an ownership modifier: If yes, then the type was provided by the programmer and is therefore accepted, and processing continues as normal. If not, there are two options: One could either report an error or use some default modifier. The implementation currently does the former.

Code This is a reference to the subtree of the abstract syntax tree which contains the definition of the symbol. Such a subtree has either a `ValDef` node (value definition) or `DefDef` node (method definition) at its root.

Unit The symbols of a compilation run may reside in different compilation units, therefore the unit where a symbol is defined is also stored. This is mainly used for error reporting.

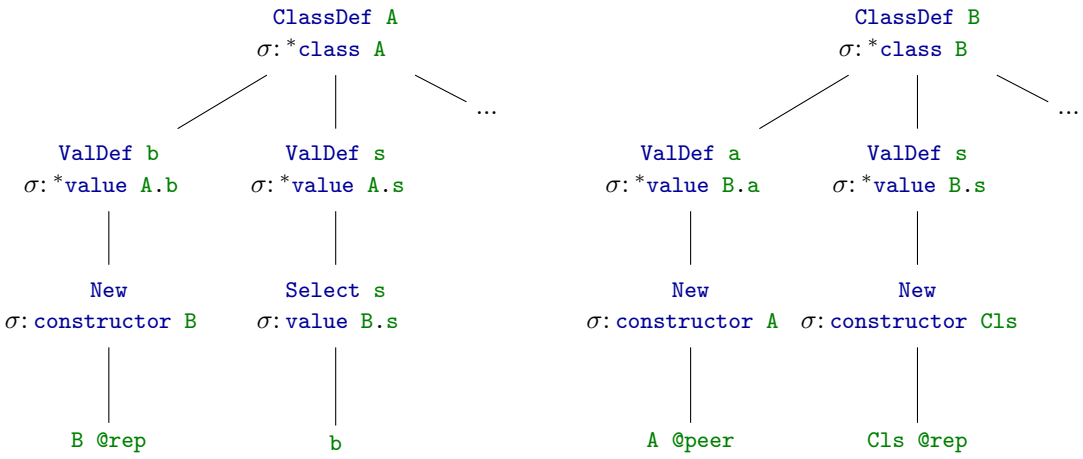
Symbol		Typed	Locked	Code	Unit
`A.b`	⇒	`false`	`false`	`ValDef b...`	`A.scala`
`A.s`	⇒	`false`	`false`	`ValDef s...`	`A.scala`
`B.a`	⇒	`false`	`false`	`ValDef a...`	`B.scala`
`B.s`	⇒	`false`	`false`	`ValDef s...`	`B.scala`

Table 4.2: Simplified state table for the classes in Listing 4.3 and Listing 4.4.

4.2.3 Inference and Propagation

The second traversal of the abstract syntax tree does the main work: It infers the ownership modifiers where necessary, propagates the ownership modifiers throughout the tree and enforces the type rules. Figure 4.2 contains highly condensed versions of the abstract syntax trees for the classes in Listing 4.3 and Listing 4.4.

The trees are processed using a depth-first traversal starting with the abstract syntax tree of the first compilation unit, in this case `A.scala`. Whenever a `ValDef` or `DefDef` node is encountered on



(a) Class `A` of unit `A.scala` from Listing 4.3

(b) Class `B` of unit `B.scala` from Listing 4.4

Figure 4.2: Simplified and condensed abstract syntax trees.

the way down to a leaf of the tree, the symbol referenced by the node gets locked. This indicates that the symbol's definition is currently being processed.

Whenever the traversal reaches a leaf node of the tree, this node is immediately processed by doing the necessary type checks. The most interesting leaf node is `This`, so there usually is not much to do: In the case of `This`, the traverser only needs to add the `this` ownership modifier to the node's type. The traverser then goes upward again and possibly processes other child nodes of the leaf's parent ρ. When it eventually gets back to ρ, it does the necessary type checks. All type-checking functions return a type τ themselves. This type τ is usually the type of the checked expression represented by the current node, but with added and possibly viewpoint adapted ownership modifiers. Since the type of node ρ may be required in the type check of ρ's parent node, it must be stored in ρ. This is done by copying τ's ownership modifiers to ρ's type. It is not possible to simply copy τ into ρ since the functions checking the type rules do not work with Scala's type representation. They use an abstraction as mentioned in Section 3.3 based on the `UType` class of the Universe type system plugin.

When the algorithm gets back to a node which defines a symbol, it copies the ownership modifiers to the type of the symbol instead of the type of the node. As mentioned in Section 4.1, such a node does not have a type in the first place, hence it could not be annotated anyways. The state of the symbol in the state table is also updated since the symbol is typed now and does not need to be locked anymore. The two copy operations where ownership modifiers get copied to the type of tree nodes or symbols take care of propagating the modifiers throughout the abstract syntax tree.

An interesting situation arises when processing the node `Select s` in Figure 4.2(a): This node represents the expression `b.s` and depends on the type of the symbol `value B.s` during the type check. Since the type check started within class `A`, this symbol has not been typed yet. The implemented inference algorithm solves this problem by processing symbol definitions when their type is required: In this case the traversal of the abstract syntax tree of class `A` would be suspended. Then another traverser would start to process the subtree referenced in symbol `value B.s`' state table entry. As soon as it finishes processing of `value B.s`' definition, the type of the symbol is known and therefore the traversal of class `A` may be resumed.

The above description of the inference algorithm considered only field accesses but it is equally valid for method calls. In fact, since every field access in Scala is done using a call to the respective getter or setter method, there is no difference at all.

Since subtrees of class definitions are processed on demand instead of sequentially, duplicate traversals of subtrees may occur. In the example of Figure 4.2 this would happen to the subtree

`ValDef s` of class `B`: It was processed on demand while traversing class `A` and would be processed again when type checking class `B`. In order to avoid this, the algorithm always checks if a symbol has already been typed before it starts processing its definition.

4.3 Static Universe Type Checker

The static type checker enforces the type rules of the Universe type system at compile time. As already noted in Section 4.2.3, the actual type checks are done in parallel to the ownership modifier inference.

4.3.1 Internal Type Representation

Section 3.3 introduced the abstraction from the type representation of the Scala compiler. In the Scala compiler, the abstract class `scala.tools.nsc.symtab.Types.Type` is the base class for all types. The classes which represent the different types are case classes, it is therefore possible to do pattern matches over types. This usually makes it fairly easy to deconstruct a type, to add ownership modifiers where necessary and to subsequently reassemble it into the original type decorated with ownership modifiers. There are some exceptions, though, but these can often be handled by provided utility methods.

Nonvariable Types

The `NType` class takes three parameters: An ownership modifier, the underlying Scala type and a list of `UType` instances as type arguments. Since the parameters are annotated with `val`, they become final but public fields of the class. It is therefore not possible to modify a `NType` instance which avoids aliasing problems in operations such as the viewpoint adaptation. This implies however that class methods like the substitution of ownership modifiers cannot modify the instance either. An obvious solution is to return new `NType` instances each time a modification is required. This works because all information about a type is stored in the three class parameters.

Type Variables

Type variables represented by class `TVarID` have only one class parameter which is the underlying type from Scala. `TVarID` offers a method `ubgamma: UType` which returns the upper bound of the underlying Scala type as a `UType` instance.

Type Errors

Instead of throwing exceptions during the type check, a special abstract `ErroneousType` class is used to indicate type errors. Some type checking methods may detect more than one error, hence there are two concrete implementations of the `ErroneousType` class: `InvalidType` contains the textual reason and optionally the position of the error while `InvalidTypes` is merely a container for several `InvalidType` instances. Both classes implement a `log` method which is used to inform the compiler about the error and to print the error message.

Since `ErroneousType` is a subtype of `UType`, it can be returned whenever a `UType` is expected, therefore the caller of a type checking method must test the result for an error.

4.3.2 Type Checks

Actual type checks are initiated during the traversal of the abstract syntax tree which takes place in the `StaticComponent` class. The traversal is done using a `scala.tools.nsc.ast.Trees.Traverser` instance similar to the one in Listing 4.2, but via a depth-first traversal. Of course the `traverse(tree: Tree)` method is more complex than the one of the example.

It consists of two sequential pattern matches: The first one locks the symbol referenced by the root node of the subtree if the root node is an unprocessed `ValDef` or `DefDef` node. This is described in Section 4.2.3 which also briefly mentions how duplicate traversals of subtrees are avoided: A subsequent pattern match checks if the current subtree's root node is an unprocessed `ValDef` or `DefDef`. In the case that it is as yet unprocessed, a pattern match over all interesting kinds of nodes of an abstract syntax tree follows.

When the root node of the current subtree matches a given pattern, the node gets processed. In general, this proceeds as follows: First the required types for a type check are retrieved. They are usually fetched from the symbol associated with the node, the subtrees of the node, the node itself, or several of these sources. Then the type checking method of the `TypeRules` implementation which corresponds to the kind of the node is called with all the required information. This involves the retrieved types and possibly also a symbol related to the node. Some of the methods also take a parameter for the position of certain constructs in order to achieve a more fine-grained error reporting – this is for example the case for the `checkMatch` method which is used to type check a pattern match: It gets the position of each `case` statement.

All type checking methods of `TypeRules` implementations return `UType` instances. Because of the error handling mentioned above, this `UType` instance could be an `ErroneousType`, so when processing the result of the type check this must be taken care of by emitting an error message if necessary. If there was no error, the returned type's ownership modifiers are usually copied to the type of the node or the referenced symbol, as mentioned in Section 4.2.3.

Nested Classes and Methods

The implemented type checker also supports nested classes and methods. However, the type rules from Section 1.2.1 do not handle either of those language features. The current implementation is therefore straightforward: Inner classes are processed like normal classes and nested methods like methods called on the receiver `this`. At least for inner classes this is certainly not optimal because of the outer `this` of an instance of an inner class: Its type currently defaults to a `peer` type, whereas `any` would certainly be better. Extending the type system with support for nested classes and methods therefore remains a subject of future work.

4.4 Runtime Checks

The implementation of the runtime checks is basically a port of the earlier work in [26] to Scala. It is interoperable with this earlier implementation for Java in that it can generate checks which use the old runtime support library, as mentioned in Section 3.4.

4.4.1 Modification of the Abstract Syntax Tree

All modifications of the AST are performed in implementations of the `RuntimeCheckTransformBase` trait: The `RuntimeCheckTransform` class creates code which targets the Scala implementation of the runtime support library while the `MJRuntimeCheckTransform` class results in code for the MultiJava implementation.

The `RuntimeCheckTransformBase` trait provides an abstract inner `RTCTransformer` class which specializes the abstract `scala.tools.nsc.ast.Trees.Transformer` class. This inner class is used for the actual transformation and implements a `transform(tree: Tree): Tree` function for the transformation. There are three kinds of subtrees which must be modified in order to add the runtime checks:

Definitions of constructor methods The body of a constructor method must be modified such that the owner of the new object gets set before the real method body is executed. This is taken care of by the `processConstructor` function which replaces the subtree of the constructor method's body by an extended one.

Calls to constructor methods The modified constructor methods need a reference to the object which contained the call to the constructor method in order to set the owner of the new object. Hence, this reference must be stored before executing the call. This is the task of the `processConstructorCall` function.

Type casts and instanceof tests These two operations are handled by the `processIsAsInstanceOf` function which makes them Universe-aware by performing additional checks that take the ownership relation into account.

The above functions get the current subtree and return the modified one which subsequently replaces the original subtree.

In order to create code for a certain runtime support library, these methods are overridden in the actual implementations of the abstract `RTCTransformer` class. The specializations are contained in the `RuntimeCheckTransform` and `MJRuntimeCheckTransform` classes and create subtrees which target the Scala or MultiJava implementation, respectively, of the library.

The abstract transformer also offers several helper methods to generate new trees which for example can be used to call a method or to log a message. These trees are used as building blocks for the new or modified subtrees which are created by the above methods.

4.4.2 Runtime Library

Most of the additional work for the runtime checks is performed in a runtime support library. It provides a singleton object in the Scala version or a class with static methods in the Java version. Both variants implement a similar interface: It comprises methods to set the owner of an object and functions to check if a given object is the owner or peer of another object. The operations are called at runtime by the functions mentioned in Section 4.4.1 to conduct the additional checks and to maintain the ownership hierarchy. Details about the Java implementation can be found in [26]; the Scala implementation closely follows this implementation.

4.4.3 Storage of the Ownership Relation

Both runtime support libraries use an optimized hash table to store the ownership relation at runtime. Whenever a new object is created, it gets inserted into the hash table together with a reference to its owner.

4.5 Logging and Error Reporting

In order to get a common interface for debugging output and error logging, the `UTSLogger` trait was devised. It is inspired by and is partially based on `logger.scala` from [27]. Another source of inspiration was log4j [11] from the Apache project. An implementation of the `UTSLogger` trait therefore supports several log levels, namely `DEBUG`, `INFO`, `NOTICE`, `WARN` and `ERROR` in ascending order of severity. The purpose of distinct log levels is to allow for a specific handling of each level: Messages on level `WARN` and above are passed on to the Scala compiler which will record the warning or error and also print the excerpt of the source code where the problem occurred. Messages from lower levels are simply printed without notifying the compiler. The `UTSLogger` also allows a user to set a minimal log level: If set to a value above `DEBUG`, messages on lower levels will be discarded.

For every log level there is a corresponding method to log a message at the given level. These methods call method `log(level: Level.Value, message: => String, pos: util.Position): Unit` for the actual logging. It takes a `level` for the message, the `message` itself, and a `position`. If the `level` is greater or equal to the global minimal log level, it will handle the message.

The position is only relevant for levels above `NOTICE`, i.e. `WARN` and `ERROR`. Lower levels do not make use of it. As the position changes while traversing the abstract syntax tree, it must be kept up-to-date in order to provide useful error messages and especially the correct source code

excerpts. This is achieved by using a stack: Whenever a new subtree is entered, its position gets pushed, and as soon as the subtree is left, it gets removed again. There are certain cases though where this is not sufficient (especially `if` and `match` expressions), so there is also the possibility to explicitly specify the position of the error.

The `message`'s type `=> String` means that the parameter gets passed by reference – it is therefore not evaluated at the time of the method call, but only when the value is actually used. This avoids possibly expensive operations (such as calls to the `toString` method of complex data structures) in the case where the minimal log level is higher than the current `message`'s level.

Each of the two plugins may use a different log level which can be set using the command line option `-P:<plugin-name>:loglevel=<level>` and a `<level>` of `debug`, `info`, `notice`, `warn`, or `error`. The default log level for both plugins is `notice`.

4.5.1 Logging and Error Reporting in the Scala Compiler

The Scala compiler already provides functionality for logging, but as it is distributed over several classes, it nevertheless seemed sensible to provide a common interface for the Universe type system plugins.

One of these classes is class `Global`, an instance of which is available through the `global` reference (see Section 1.1.6). It basically provides four methods for the reporting of messages: `inform(msg: String)`, `log(msg: AnyRef)`, `warning(msg: String)` and `error(msg: String)`. `log` will only print a message if logging is activated for the current compiler phase – if logging is enabled, it will print a message to the standard error output using the `inform` method. The `warning` and `error` methods use an instance of `scala.tools.nsc.reporters.Reporter`, but without passing the position of where the warning or error was encountered.

Compilation units as represented by `scala.tools.nsc.CompilationUnits.CompilationUnit` also provide methods for warnings and error reporting which actually make use of the position of the error. This results in warnings and error messages with short excerpts of the source code at the given position. These two methods are used by the `UTSLogger` which therefore must be kept up-to-date with the current compilation unit during the traversal of the abstract syntax trees. This is achieved by updating the `UTSLogger`'s reference to the current unit whenever the unit changes.

4.6 Testing

Scala provides the SUnit[1] framework for unit testing which is in the same spirit as JUnit for Java. A SUnit test is a Scala application which executes actual `scala.testing.SUnit.TestCase` implementations. SUnit is still very elementary when compared to the well-known JUnit testing framework, but it serves its purpose.

Devising a testing strategy for the compiler plugins turned out to be not entirely trivial – the main problem is that the classes and methods in the plugins work with abstract syntax trees and the compiler's type representation. These data structures would have been very tedious to create by hand. Therefore the tests of the plugins take a Scala source code file and try to compile it using the compiler plugins. Since the number of expected errors and warnings is known to the test case, a comparison of actual and expected numbers allows to conclude if the test was successful or not. As one plugin adds runtime tests to the programs, they must also be executed to see if they yield the correct result. This second test is fairly similar in that it simply tests if the program terminates without error or if an exception is thrown during execution.

The tests use several auxiliary classes, especially two factories which produce test cases where a given input program gets compiled or executed. A `TestRunner` trait with a `main` method is mixed into those classes which provide a test suite in order to actually execute the tests. It also collects statistics and finally prints the number of executed tests together with the number of failures. If there were any failures, it terminates the program with a non-zero exit status.

[1] Short for "Scala Unit", which must not be mistaken for the SUnit unit testing framework of Smalltalk, which is the original source of the xUnit design.

Although the testing procedure is very simple, it proved to be useful during the development of the plugins. Whenever a new feature of Scala was to be implemented or an error was found, a new and short example program for the test-suite could be written. It was also useful for regression testing, especially when refactoring the plugins.

A quick glance at the Scala source code reveals that the test-suite for Scala itself works in a similar way: They also try to compile input programs and subsequently compare the actual output to the expected one. This is different from the aforementioned strategy which only compared the error count: The advantage of the latter is that it is not required to update any tests when some error message of the compiler changes. An obvious drawback is that one does not necessarily catch erroneous errors as long as the count matches.

4.6.1 Test Cases

Most test cases, i.e. example programs, are rather small, their length ranges from 2 to 94 lines of code, with an average of about 30 lines. They usually contain several variations of the same language construct or rather arbitrary program code which exposed a problem in the type checker. There are currently more than 55 examples to test the static type checker and more than 5 for the plugin which adds runtime checks.

Two of the examples implement simple containers: A map and a stack. The map is a port of the example in [6] to Scala. It contains an additional `each(fun: (K,V)=> unit)` method which takes a first-class function `fun` and applies it to every entry of the map.

The test cases should cover all language constructs supported by the implementation of the type system. However, real programs are often fairly different from synthetic examples. It was therefore tried to annotate and compile a subset of Scala's collection library. This actually exposed several errors in the type checker which subsequently could be fixed. Unfortunately, the collection library uses as yet unsupported variance annotations quite often. In the end, there was not enough time left to extend the type checker or to experiment with other, similarly large programs. But the experiments with real program code at least made the implementation more robust.

5 Conclusion

This chapter starts by presenting the status of the implementation and subsequently mentions some experiences with the development of a compiler plugin for Scala. It closes with a list of possible extensions of the presented implementation and some related ideas for future work.

5.1 Status of the Implementation

The implementation works, but it does not yet support all language constructs of Scala. However, a superset of the minimal language from [6] and the rules from Section 1.2.1 are implemented. This comprises the following:

- Classes and inheritance.
- Instance fields, dynamically bound methods, and constructors.
- Usual operations on objects: Allocation, field read and update, and type casts.
- Control structures such as *if-then-else* and *match*. As the Scala compiler converts loops to iterations over collections or to recursive methods, *while* and *for* loops are also supported.

Some other language constructs are at least partially supported, but not thoroughly tested:

- Anonymous classes: They are supported by the static type checks, but currently unsupported by the plugin for the runtime checks. This is at least partially caused by a bug in the Scala compiler [31].
- Nested classes and methods are basically supported, but with the restrictions from Section 4.3.2: The implemented type rules are certainly not optimal.
- Traits and mixin-composition: These can be handled like classes and inheritance, but this has not been tested very well.
- Exceptions: *try-catch-finally* expressions are basically supported, but the main modifier of exceptions does not yet default to `any`, which is in opposition to Section 1.2.1.
- Singleton objects: These are basically supported, but the support is not very well tested, especially not at runtime.
- First-class functions: The Scala compiler converts first-class functions to `FunctionN` specializations, they can therefore be handled very much like normal classes and objects. However, it is usually necessary to declare the parameter types when defining an anonymous first-class function, which is in most cases not required in Scala.

Some constructs, however, are as yet completely untested and therefore possibly also completely unsupported[1]. The most notable ones are the following:

- Self-type annotations.
- Variance annotations of type parameters: Earlier work [27] already contains type rules which support variance annotations, but this has not been implemented.

[1] As mentiond in Section 4.1, the AST is a desugared version of Scala. It is therefore possible that a not explicitly supported source language construct gets converted to a construct in the AST which is already supported.

Ownership modifier inference generally works, but it is only implemented for variable types and method return types. It is as yet unsupported for type arguments of calls to polymorphic methods and instantiations of generic classes. Hence, it is necessary to provide more type annotations than in a normal Scala program where these are often inferred.

5.2 Development of a Compiler Plugin for Scala

Developing a compiler plugin for the Scala compiler is relatively straightforward. It may take quite some time though to get an overview of the Scala compiler itself as there does not seem to be a high-level documentation of the compiler. This problem gets emphasized by the fact that there is often only little documentation in the compiler's source code.

Scala itself, on the other hand, is documented very well. It is therefore quite easy to start development with Scala. The implementation of the compiler also provides many facilities which may be used in a plugin, but it can be difficult to actually find them.

Summarizing, one could say that programming in Scala makes up for quite some of the deficiencies of the compiler's documentation. It is also relatively easy to actually extend the Scala compiler using plugins once the inner workings and data structures of the compiler are more or less understood.

5.3 Future Work

Support for a broader or even the complete set of Scala's language constructs would certainly make the implementation more useful. This would also open a new set of possibilities: One could then annotate and recompile the complete class library of Scala. The annotation overhead should be reasonable because of the defaulting and the local ownership modifier inference. However, it would be useful if the Scala compiler supported external annotation files similar to JML [13] as this would allow a separation of the annotations from the implementation.

The additional runtime checks could also be changed if the complete class library was recompiled using the plugins: This would make it possible to mix a trait with an owner field into every compiled class which would make the maintenance of a hash table dispensable in many cases.

Runtime checks currently consider only the main modifier of a type in casts and instanceof tests. This limitation is imposed by Java and its lack of support for runtime types in type arguments. However, [6] also provides a runtime model for GUT which was implemented for the MultiJava compiler [23]. It should be possible to implement this for Scala, too.

The imposed annotation overhead is already relatively small. In order to further reduce the overhead, the as yet missing inference of type arguments should also be implemented. One could also come up with a more sophisticated defaulting strategy.

It would also be possible to implement static and runtime inference as it was done for Java in [14, 12, 2] for Scala. Static inference could also use the abstract syntax tree and should work similarly as for Java. Runtime inference would possibly be more difficult, though, as the compilation of Scala code results in byte code which may look very different from the input program. Mapping Java byte code to a Java program, for instance, is more straightforward than mapping Java byte code to the corresponding Scala program.

Apache Ant [9] is slowly becoming slightly old-fashioned while Apache Maven [33] is gaining momentum. Switching to Maven and possibly extending its Scala plugin [3] to support compiler plugins would have at least two advantages: (1) The implementation of the Universe type system has become quite large, but it contains several rather independent parts (plugins, default type rules, runtime libraries, annotations, and some common classes). Maven would make it easy to spread the implementation over several sub-projects. (2) The dependency management could be handed to Maven. This would generally ease the installation and use of compiler plugins.

The plugin which adds runtime checks to the abstract syntax tree depends on the first plugin which infers and propagates ownership modifiers. This dependency is not very explicit: It is only

visible in the `runsAfter` field of the second plugin. However, the second plugin could also run after a later, built-in phase which would make the dependency invisible. One can currently pass the `-Xplugin-require:<plugin>` option to `scalac` which makes the compiler abort the compilation if a given plugin is not available. It would be useful if this could also be specified explicitly in a plugin which depends on another plugin.

A Developer's Guide for the Universe Type System Plugins

A.1 Introduction

This guide highlights some technical details which may be useful when extending or modifying the Universe type system implementation for Scala. Most of these details were not or only briefly covered in the main report [30] which is nevertheless recommended reading.

In order to use and develop the plugins, a working installation of Scala 2.6.1 is recommended. The plugins were developed and tested with Scala 2.6.1, but some experiments with pre-release versions of Scala $>$ 2.6.1 looked promising. Since the plugins make use of Java's generics, Java $\geq$ 1.5.0 is also needed, as well as Apache Ant $\geq$ 1.6.3. It is further assumed that the environment variable `$SCALA_HOME` refers to the directory where the Scala installation resides.

A.2 Directory Layout

The directory layout for the plugin sources was adopted from Apache Maven [33], even though the build system for the plugins employs Apache Ant [9]. Maven uses a standard directory layout which separates the implementation from the tests and generally follows best practices. Listing A.1 shows the actual layout of the tree.

The `src` directory contains the source code and other resources like the descriptors for the plugins. When `ant` is executed to build the project, it will save the resulting files, i.e. class files, Java archives, and the `sbaz` packages to the `target` directory.

Tests for the implementation are stored in the `src/test` subdirectory. There is a test suite for each plugin, a test for the hash table used in the Scala implementation of the runtime support library, and a test which checks if the inferred annotations are stored persistently in the class files. The test suites for the plugins use the example programs below `src/examples` as input.

Ant uses the file `build.xml` as a build script. It utilizes several properties which can be overridden by specifying them in the file `build.properties`. See Section A.5 for some of the most interesting properties.

A.3 Extending the Type Checker

There are two ways to customise or extend the type checker: One could provide a custom implementation of the defaults or one could write a specific implementation of the actual type rules. The former modifies the behaviour of the default implementation of the type rules whereas the latter is more powerful and flexible, but obviously also more involved.

Figure A.1 and Figure A.2 show the classes and traits which are concerned with the modification of the type checker's defaults and the customization of its type rules, respectively.

A.3.1 Defaults

The trait `UTSDefaults` defines some default values. There are three kinds of defaults:

Default ownership modifiers These modifiers are used where they were neither specified nor inferable in a program. This is usually the case for all members of classes which were not compiled using the Universe type system plugins. It also applies to object creation if no

```
.
|-- build.properties
|-- build.xml
|-- src
|   |-- examples
|   |   `-- scala
|   |       `-- ch
|   |           `-- ...
|   |-- main
|   |   |-- resources
|   |   |   `-- plugin
|   |   |       |-- runtimecheck
|   |   |       |   |-- scalac-plugin.xml
|   |   |       |   `-- uts-runtime.version.properties
|   |   |       `-- staticcheck
|   |   |           |-- scalac-plugin.xml
|   |   |           `-- uts-static.version.properties
|   |   `-- scala
|   |       `-- ch
|   |           `-- ...
|   `-- test
|       `-- scala
|           `-- ch
|               `-- ...
`-- target
    `-- ...
```

Listing A.1: Directory layout of the plugin sources.

modifier was specified, like in `new AnyRef`. There are default modifiers for five cases: (1) Upper bounds of type variables, where it is `any`. (2) Immutable types like `Int`, `String`, etc., where it is also `any`. (3) Singleton objects which are defined inside of a class (`peer`) or (4) in a package (`any`). (5) A general default of `peer`. These defaults are defined in the `StandardDefaults` class and may be overridden.

Pure methods and immutable types Since the Scala library does not declare pure method as actually being pure, a way to denote methods without `@pure` annotation as pure methods was devised. This was necessary as some type rules allow only calls to pure methods on certain references. It basically works by assuming that all methods of value types (`scala.AnyVal` instances) are pure. Since there are also other types which may offer pure methods, this can be customised: The list `immutableTypes` contains certain other types like `java.lang.String`. In addition, the map `pureMethods` maps class names to lists of pure methods of the respective class. In order to cover inheritance, classes in the map are interpreted as base classes and it is assumed that methods in subclasses which override any method of the list are also pure.

Default classes There are three values for default classes: Two of them are used to provide the names of the two main objects of the runtime support libraries. These objects are used to store the ownership relation at runtime. The third class name specifies the default implementation of the type rules which should be used in the static type check.

To simplify access to the defaults, there is the static `object UTSDefaults` which implements the `UTSDefaults` interface and encapsulates an actual instance of the `UTSDefaults` trait. Every method of the `UTSDefaults` interface which is called on the static object is forwarded to the encapsulated instance. The instance gets initialised from a user-selected class if the `-P:<plugin name>:defaults=<class>` command line option for the plugin was given. `<plugin name>` is ei-

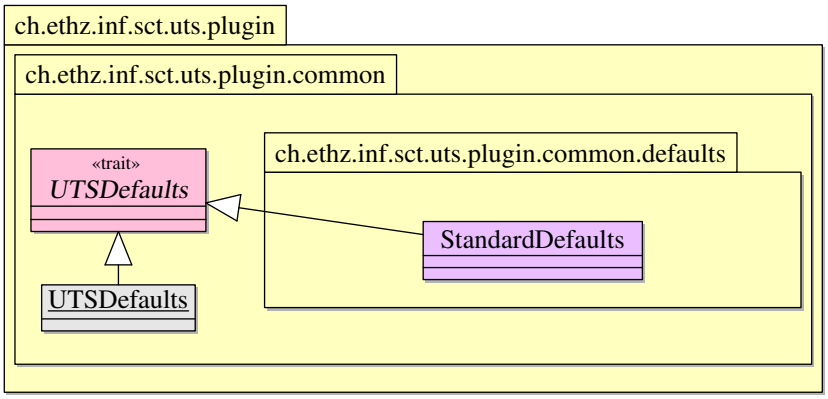


Figure A.1: Defaults for the type checker.

ther `uts-static` for the plugin which performs the static type checks or `uts-runtime` for the plugin which adds the runtime checks. It is therefore possible to specify different defaults for the plugins although this presumably does not make a lot of sense. Hence, it is recommended to specify custom defaults for the `uts-static` plugin only. These will automatically be used in the `uts-runtime` plugin which runs after the one for the static checks.

A.3.2 Type Rules

A customised variant or a different version of the type rules must provide an actual `TypeRules` implementation. The implementation which should be used during the type check can be selected by supplying the `-P:uts-static:typerules=<actual TypeRules implementation>` option to `scalac`. As mentioned in Section A.3.1, this could also be achieved by providing a custom `UTSDefaults` implementation which defines a different default for the type rules implementation. This might be required anyway, as a different implementation might need other defaults for the ownership modifiers.

Figure A.2 shows the main traits, one class and one abstract class which are involved in the implementation of type rules and the abstraction from the Scala compiler's type representation. It contains only a condensed version of the class hierarchy, the main report in [30] also displays the abstract inner classes of the traits and several auxiliary classes.

The `ch.ethz.inf.sct.uts.plugin.staticcheck.common` package at the top provides an abstraction from the Scala compiler's type representation. It gets assembled in the `TypeAbstraction` trait by mixing in traits which provide abstract factory methods and abstract inner classes that are used for the abstraction. The `ch.ethz.inf.sct.uts.plugin.staticcheck.rules.default` package at the bottom provides the default implementation of the type rules. Its traits specialize traits from the package at the top and provide concrete implementations of their abstract inner classes. In addition, they also implement the factory methods in order to create instances of the inner classes.

It is recommended to look at the default implementation of the type rules before implementing a different variant of the rules. The default implementation consists of several traits and the concrete `DefaultTypeRules` class which implements the actual rules. Their signatures are declared in the abstract `TypeRules` class. All those traits which are used to abstract from the Scala compiler's type representation are mixed into the `DefaultTypeAbstraction` trait which also serves as the self type of these traits (indicated by the black diamonds). In addition, it mixes in the `DefaultOwnershipModifierExtender` trait which provides a specialized implementation of the behavior of ownership modifiers which also depends on the variant of the implemented type rules.

The architecture of the abstraction from the compiler's type representation uses Scala's self types and mixin class composition. This results in a set of components which should be reusable in different implementations of the type rules. It is also similar to the design of the Scala compiler itself which is explained in a case study of [22].

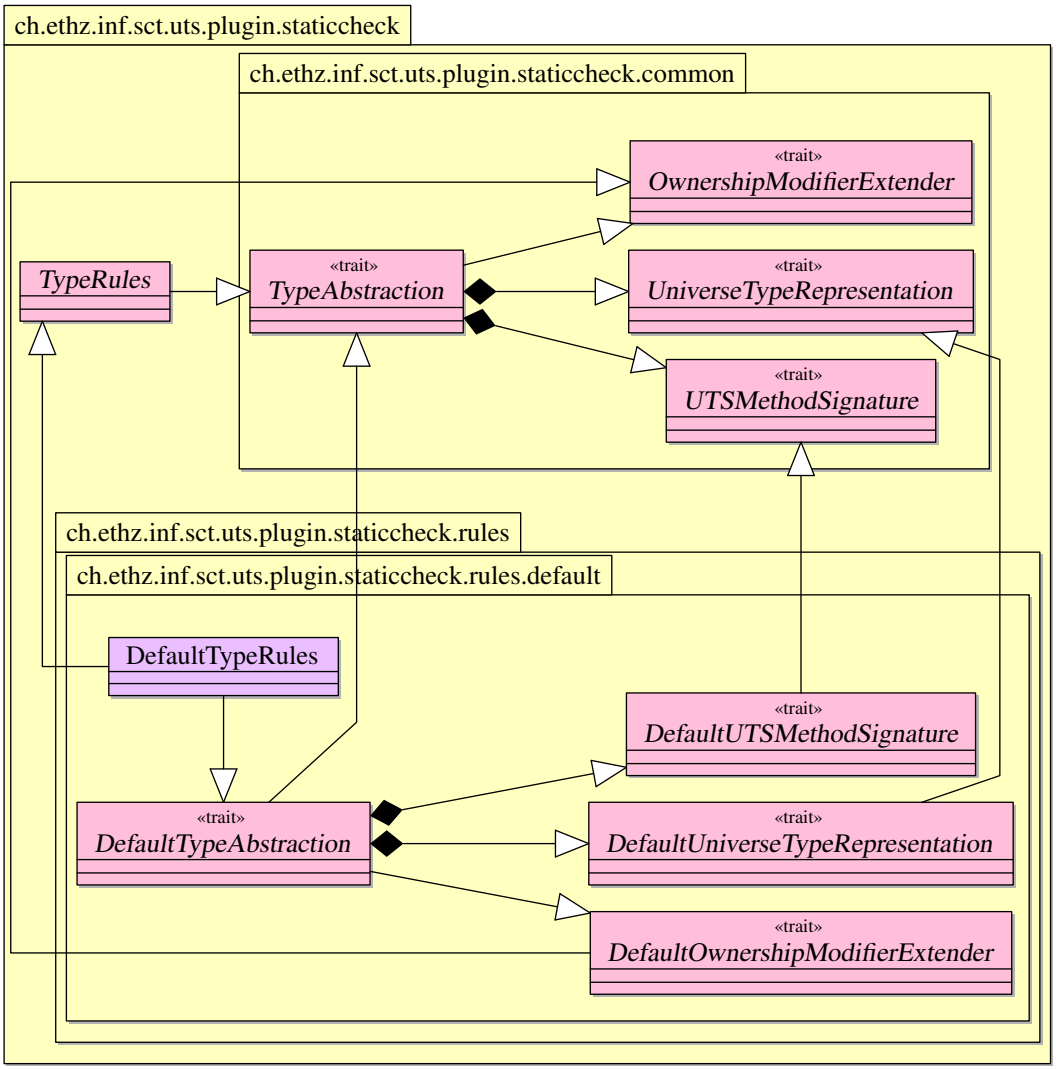


Figure A.2: Implementation of type rules.

Type Checking Methods

A concrete implementation of the abstract `TypeRules` class must implement those methods which are called during the type check. These methods take one or more types or compiler symbols as arguments and check if the respective type or well-formedness rules hold. They return the viewpoint adapted type which corresponds to the kind of check they did (e.g. the check of a method call would return the method's viewpoint adapted return type) or a special `ErroneousType` which denotes a type error. If the check should print a warning instead of an error, this must be done in these methods as well since all `ErroneousType` instances returned to the caller are handled as errors. The warning can be generated using the `logger.warn(=> String)` method which will also print a short source code excerpt of the current position. Debugging output can be produced using `logger.debug(=> String)`, `logger.info(=> String)` and `logger.notice(=> String)`. The output of the `logger` may be suppressed by the default log level (which is `notice`) or the level which was set using `-P:<plugin name>:loglevel=<level>` when invoking the compiler.

Some type rules may employ common helper methods which are for example used to get the type of a field or method. The type they return is usually also viewpoint adapted. Since these methods may depend on the variant of the type system which is implemented, they are declared

as abstract methods in the abstract `TypeRules` class and must therefore be provided as well.

Types

The `UniverseTypeRepresentation` trait provides abstract `NType` and `TVarID` classes which are used in the abstraction from the compiler's types. They contain some abstract methods, for example for the subtype and well-formedness checks, which must be implemented in a concrete implementation of the type rules.

A.4 Customizing the Runtime Checks

The behavior of the runtime checks can also be customized. Both implementations (the one in Scala and the one in Java) of the runtime support library are either based on or taken from earlier work [26]. This guide will therefore not go into the details of this customization.

Both libraries are divided into implementation and policy classes. At runtime, one of each kind is needed, and it is possible to select the class by defining a property. The implementation class maintains the ownership relation and it also does the additional checks. A policy class defines the behavior of the runtime checks, for example if invalid casts should generate an exception or only result in a warning.

A.5 Compilation

As mentioned before, the build system for the Universe type system plugins is based on Apache Ant. A list of available targets and their description may be obtained by calling `ant -p`. The default `build` target simply builds the plugins and places the resulting class files and Java archives in the `target` directory. The target `run.tests` can be used to conduct some tests: Most of them will compile a given input program which contains errors and compare the error count to the expected number of errors. If the numbers match, the test was successful.

If desired, the build may be customized to some degree without changing the `build.xml` file. This may be achieved by editing the `build.properties` file which gets loaded in the build script. The most interesting properties used in `build.xml` are the following:

`version` The version of the project is used for the `sbaz` packages and in the title of the API documentation.

`package.name.prefix` Prefix which is used for the name of the Java archive files and the `sbaz` packages.

`package.urlbase` URL of the directory on the web server where the `sbaz` package of the plugins will be available for download. The name of the `sbaz` package gets prefixed by this property's value and the resulting URL is then stored in the advertisement file for the Scala Bazaar.

`scala.home` The directory containing the Scala distribution which should be used during the build. If neither this property is set in `build.properties` nor `$SCALA_HOME` was defined, the build script will use `~/sbaz` as the default value for `scala.home`.

A.5.1 Building a Distribution

The Ant build script supports a `dist` target which will create a Java archive file of the plugin's source, `sbaz` packages and `sbaz` advertisement files to share the packages using a Scala Bazaar. Section A.6.4 describes how the latter can be achieved. The packages contain the plugins, the annotations, the runtime libraries for programs which use the Universe type system, the source code of the plugins, and the API documentation.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <servlet>
    <servlet-name>uts-scala</servlet-name>
    <servlet-class>sbaz.Servlet</servlet-class>
    <init-param>
      <param-name>dirname</param-name>
      <param-value><!-- path to Bazaar directory --></param-value>
    </init-param>
    <load-on-startup>2</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>uts-scala</servlet-name>
    <url-pattern>/uts-scala</url-pattern>
  </servlet-mapping>
</web-app>
```

Listing A.2: `web.xml` for the `sbaz` servlet.

A.6 Setting Up a Scala Bazaar

Scala Bazaars [28], or `sbaz` for short, is a package management system for Scala. It consists of two parts: The `sbaz` command line tool and a corresponding Java servlet for the server side. A universe in `sbaz` is basically a list of packages.

The server side of `sbaz` requires a servlet container such as Jetty [4] (which was used in this guide) or Apache Tomcat [10] to run. There are several ways for deploying servlets to these containers, but this guide will only describe the deployment using a *Web Application Archive* (WAR) file.

A.6.1 Creation and Deployment of `sbaz.war`

The WAR file must contain the file `sbaz.jar` and usually also the file `scala-library.jar`. Both files can be found in the official Scala distribution in the `share/scala/lib` directory.

1. Create the required directory layout: `mkdir -p sbaz-servlet/WEB-INF/lib`
2. Copy the files `sbaz.jar` and `scala-library.jar` into `sbaz-servlet/WEB-INF/lib`.
3. Create the file `sbaz-servlet/WEB-INF/web.xml` according to Listing A.2 and change the `param-value` to a directory on the server where the servlet container has read-write access. This will be the directory where the configuration of the universe and its package descriptions are going to be stored. It will be referred to as *configuration directory* in the remainder of this document. Since this directory may contain sensitive configuration information, it is highly recommended to use a directory which is not publicly accessible.
4. Create the WAR file: `jar cvf sbaz.war -C sbaz-servlet .`

Now that the `sbaz.war` file has been created, it may be copied to the `webapps` directory of the servlet container. Depending on the container and its configuration, it might be necessary to restart the servlet container in order to load the servlet.

```
<simpleuniverse>
    <name>uts-scala</name>
    <description>
        The Scala Bazaar for the Universe type system plugins.
    </description>
    <location><!-- Servlet URL --></location>
</simpleuniverse>
```

Listing A.3: Bazaar descriptor for the Universe type system plugins, to be used on the server.

```
<req>
  <read/>
  <editkeys/>
</req>
```

Listing A.4: `keylessRequests` for setup.

```
<req>
  <read/>
</req>
```

Listing A.5: `keylessRequests` for use.

A.6.2 Setup of the Bazaar

Listing A.3 shows a Bazaar descriptor which could be used for the Universe type system plugins. It must be stored in a file called `universe` in the configuration directory which was set in the `web.xml` file. The `location` must contain the URL of the servlet, which would be `http://<name of the server>:<port>/sbaz/uts-scala` if one followed this guide.

It should now be possible to access the URL of the servlet using a web browser and to get back the Bazaar descriptor together with a list of available packages, which is currently still empty.

A.6.3 Setup of Access Control

The default configuration of the Bazaar is quite restrictive, it is therefore neither possible to get a list of available packages nor to upload new packages using the `sbaz` command.

In order to setup the key-based access control using `sbaz`, the default restrictions must be relaxed. This can be done by placing a file called `keylessRequests` in the configuration directory. It should contain a configuration like the one in Listing A.4 which allows the retrieval of the package list and key management. For details about the contents of the file, see [29]. It can and should be replaced by a more restrictive configuration like the one in Listing A.5 after setting up access control. This still allows the retrieval of package lists without any authentication, but prevents unauthorised key management. After changes to the `keylessRequests` file, the servlet container must be restarted.

Setup of Access Control using `sbaz`

The server side is now ready for the actual access control setup. It is therefore time to make `sbaz` use the new universe: This is done using `sbaz setuniverse <path to the Bazaar descriptor file>` where the Bazaar descriptor file should look like the one in Listing A.6. It refers to the Scala Bazaar with the Universe type system plugins and the `scala-dev` Bazaar with Scala itself. The first universe in the descriptor file is the one `sbaz` subsequently interacts with, later ones are only used to retrieve package lists. If the command results in a `java.io.FileNotFoundException`, the local *managed directory*[1] where `sbaz` puts everything it installs presumably has not yet been set up. This can be done by executing `sbaz setup`.

`sbaz showuniverse` should now contain the freshly setup universe as well as the `scala-dev` universe. `sbaz available` should print a nonempty package list with the packages from the

[1] This is usually either the directory `$SCALA_HOME` or `~/sbaz`.

```
<overrideuniverse>
  <components>
    <simpleuniverse>
      <name>uts-scala</name>
      <location><!-- URL to the uts-scala Scala bazaar --></location>
    </simpleuniverse>
    <simpleuniverse>
      <name>scala-dev</name>
      <location>http://scala-webapps.epfl.ch/sbaz/scala-dev</location>
    </simpleuniverse>
  </components>
</overrideuniverse>
```

Listing A.6: Bazaar descriptor for the `sbaz` tool.

`scala-dev` universe. One may now create a first key by executing `sbaz keycreate $USER '<edit nameregex=".*"/>'`. The string `<edit nameregex=".*"/>` is a so-called *message pattern*. `edit` means that someone presenting the correct key may upload new packages if the package's name matches the `nameregex`. The execution of the command will create the file `keyring` in the configuration directory and the file `keyring.uts-scala` in the `meta` subdirectory of the `sbaz` managed directory. The two automatically generated files contain common secrets, the keys, which are used for authorisation. `sbaz keycreate $USER '<editkeys/>'` will create another key which can be used for key management after removing the corresponding option from the `keylessRequests` file.

A.6.4 Uploading a First Package

Before uploading a package, care must be taken that `sbaz` uses the correct Bazaar, otherwise the upload might yield unexpected results. This can be done by executing `sbaz showuniverse`: The first universe in the list must be the one which should receive the package advertisements. If this is not the case, the universe must be set according to Section A.6.3.

The key for the message pattern `<edit nameregex=".*"/>` created in Section A.6.3 allows the upload of package advertisements using `sbaz share <package.advert>`. This sends the advertisement to the servlet which stores it in the file `packages` of its configuration directory. Calling `sbaz available` afterwards should yield the non-empty list of currently shared packages. The command `sbaz retract <package>/<version>` allows to remove a certain package from the universe.

Since the package advertisement does not contain the actual package, only its URL, the `sbaz` package must be uploaded to the given URL by other means. This could for example be done using `scp`.

In order to simplify the upload and the retraction of package descriptions, the Ant build script provided with the Universe type system plugins also supports a `share` and a `retract` target.

List of Figures

List of Listings

Bibliography

[1] Philippe Altherr. *A typed intermediate language and algorithms for compiling Scala by successive rewritings*. PhD thesis, École Polytechnique Fédérale de Lausanne, 2006.

[2] Marco Bär. Practical Runtime Universe Type Inference. Master's thesis, Swiss Federal Institute of Technology Zurich (ETHZ), Department of Computer Science, 2005–2006.

[3] David Bernard et al. Apache Maven Plugin for Scala. `http://www.scala-tools.org/mvnsites/maven-scala-plugin/`.

[4] Mort Bay Consulting. Jetty. `http://www.mortbay.org/`.

[5] Vincent Cremet, François Garillot, Sergueï Lenglet, and Martin Odersky. A Core Calculus for Scala Type Checking. In *Proc. MFCS*, Springer LNCS, September 2006.

[6] Werner Dietl, Sophia Drossopoulou, and Peter Müller. Generic Universe Types. In E. Ernst, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 4609 of *Lecture Notes in Computer Science*, pages 28–53. Springer-Verlag, 2007.

[7] Werner Dietl and Peter Müller. Universes: Lightweight Ownership for JML. *Journal of Object Technology*, 4(8):5–32, 2005. `http://www.jot.fm/issues/issue_2005_10/article1`.

[8] Werner Dietl, Peter Müller, and Daniel Schregenberger. *Universe Type System – Quick-Reference*. Swiss Federal Institute of Technology Zurich (ETHZ), Department of Computer Science, August 2005. `http://sct.ethz.ch/research/universes/tools/juts-quickref.pdf`.

[9] Apache Software Foundation. Apache Ant. `http://ant.apache.org/`.

[10] Apache Software Foundation. Apache Tomcat. `http://tomcat.apache.org/`.

[11] Apache Software Foundation. log4j. `http://logging.apache.org/`.

[12] Nathalie Kellenberger. Static Universe Type Inference. Master's thesis, Swiss Federal Institute of Technology Zurich (ETHZ), Department of Computer Science, 2005–2006.

[13] Gary T. Leavens et al. JML - the Java Modeling Language. `http://www.jmlspecs.org/`.

[14] Frank Lyner. Runtime universe type inference. Master's thesis, Swiss Federal Institute of Technology Zurich (ETHZ), Department of Computer Science, 2005.

[15] Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*. PhD thesis, Fernuniversität Hagen, 2001.

[16] Martin Odersky. Scala By Example, December 2007.

[17] Martin Odersky. The Scala Experience, 2007. OOPSLA Tutorial, `http://lamp.epfl.ch/~odersky/talks/pppj07.pdf`.

[18] Martin Odersky. The Scala Language Specification, Version 2.6, September 2007.

[19] Martin Odersky, Philippe Altherr, Vincent Cremet, Iulian Dragos, Gilles Dubochet, Burak Emir, Sean McDirmid, Stphane Micheloud, Nikolay Mihaylov, Michel Schinz, Lex Spoon, Erik Stenman, and Matthias Zenger. An Overview of the Scala Programming Language (2. edition). Technical report, École Polytechnique Fédérale de Lausanne, 2006.

[20] Martin Odersky et al. Scala. `http://www.scala-lang.org/`.

[21] Martin Odersky, Christoph Zenger, and Matthias Zenger. Colored Local Type Inference. In *POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 41–53, New York, NY, USA, 2001. ACM.

[22] Martin Odersky and Matthias Zenger. Scalable Component Abstractions. In *OOPSLA 2005*, 2005.

[23] Mathias Ottiger. Runtime Support for Generics and Transfer in Universe Types. Master's thesis, Swiss Federal Institute of Technology Zurich (ETHZ), Department of Computer Science, 2007.

[24] Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, 2000.

[25] Michel Schinz and Philipp Haller. A Scala Tutorial for Java Programmers, December 2007.

[26] Daniel Schregenberger. Runtime Checks for the Universe Type System, 2004. Semester Thesis.

[27] Daniel Schregenberger. Universe Type System for Scala. Master's thesis, Swiss Federal Institute of Technology Zurich (ETHZ), Department of Computer Science, 2006–2007.

[28] Lex Spoon. Scala Bazaars. `http://www.lexspoon.org/sbaz/`.

[29] Lex Spoon. *Scala Bazaars Manual*, March 2006. `http://www.lexspoon.org/sbaz/manual.html`.

[30] Manfred Stock. Implementing a Universe Type System for Scala. Master's thesis, Swiss Federal Institute of Technology Zurich (ETHZ), Department of Computer Science, 2007–2008.

[31] Manfred Stock. Compiler crashes when using a plugin after phase "erasure" and certain later ones, 2008. `https://lampsvn.epfl.ch/trac/scala/ticket/375`.

[32] The MultiJava Team. MultiJava. `http://multijava.sourceforge.net/`.

[33] Jason van Zyl et al. Apache Maven. `http://maven.apache.org/`.

[34] Bill Venners, Martin Odersky, and Lex Spoon. First Steps to Scala, May 2007.