User's Guide

Universe Type System for Scala

Manfred Stock

January 31, 2008

Contents

1 Introduction

The Universe Type System [3] is used to control aliasing and dependencies in object-oriented programs. Its underlying basis is the concept of ownership where each object is owned by at most one owner object. The objects are organized 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. See Figure 1 for an example of such a tree. When enforcing the so-called *owner-as-modifier* discipline, the owner must have control over the modification of its objects.

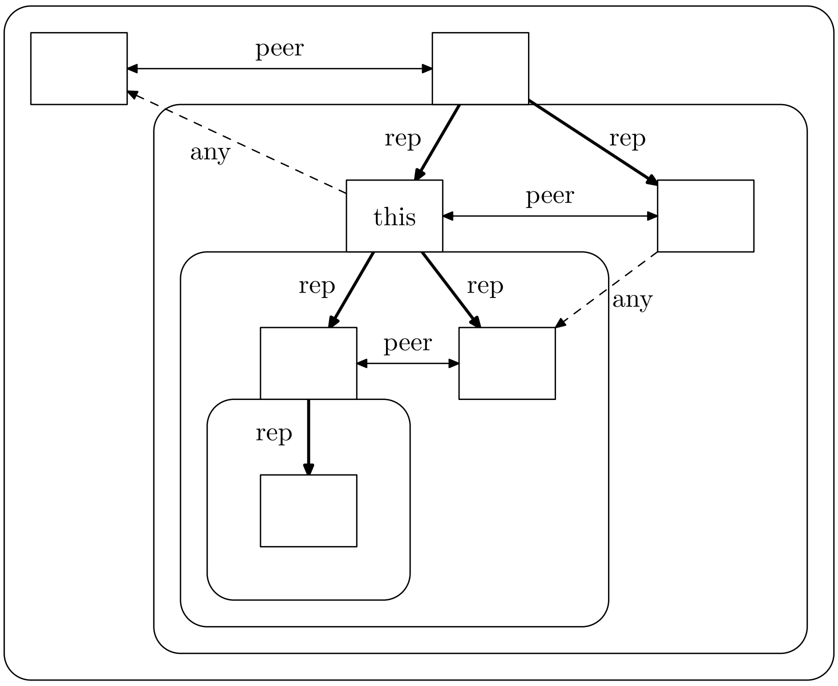

Figure 1: Ownership relations in an object structure.

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 [4].

2 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 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.

2.2 Types

Figure 2 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.

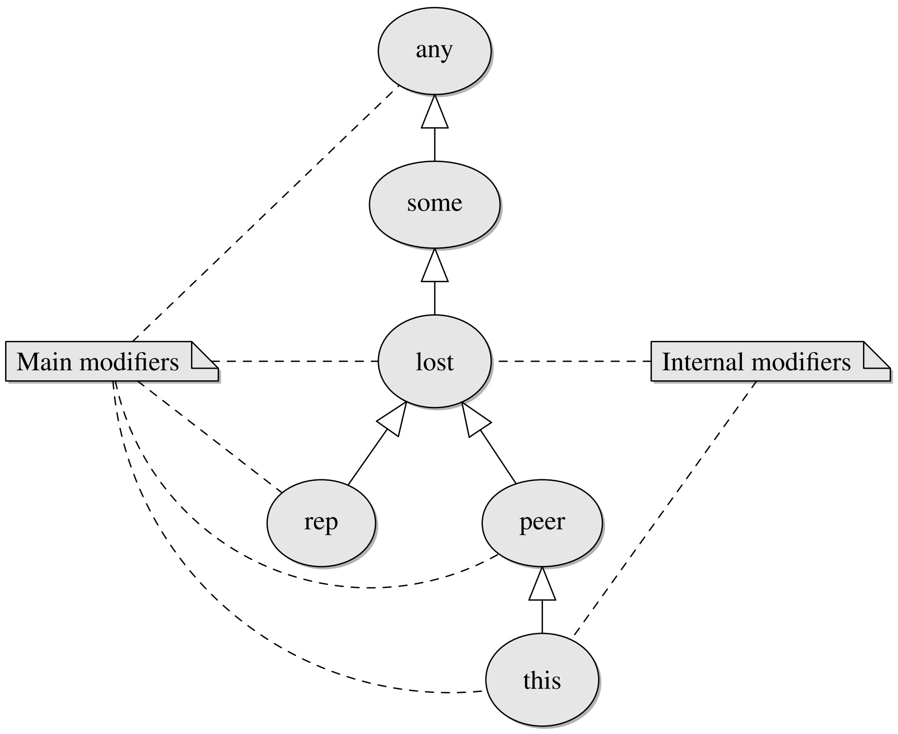

Figure 2: Subuniversing of ownership modifiers.

2.3 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.

2.4 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.

2.5 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, 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.

2.6 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 [2] 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.

2.7 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.

2.8 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.

3 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.

3.1 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.

3.2 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 [5], and part of MultiJava [7]. 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 [3, 5].

`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.

[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 1: Bazaar descriptor for `scala-devel` and the Universe type system plugins.

3.3 Scala Bazaars

Scala Bazaars [6] 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 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.

3.4 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 3.2 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.

4 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

```scala
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: Example program which uses ownership modifier annotations.

can usually either be inferred automatically or the default of `peer` is acceptable. The program can subsequently be (2) compiled and (3) run.

4.1 Annotations

As mentioned in Section 2.1 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: 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. [3, 2]) 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`.

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.

4.2 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.

4.3 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.

4.3.1 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.

4.3.2 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.

4.3.3 Options for the Addition of Runtime Checks

As already mentioned in Section 3.2, 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 [5] which for example allow a visualization of the object store.

4.4 ant

Listing 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 [8]. This should ease a future migration of a project to Maven using the Scala plugin [1] 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.

4.5 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 4 shows an Ant build script which includes the script from Listing 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
<?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 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 4: Example `run.xml` for use with the `build.xml` from Listing 3. These files can also be merged.

References

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

[2] 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.

[3] 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`.

[4] 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`.

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

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

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

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