Developer's Guide

Universe Type System for Scala

Manfred Stock

January 31, 2008

Contents

1 Introduction

The Universe Type System [2] 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.

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 [9] which is nevertheless recommended reading.

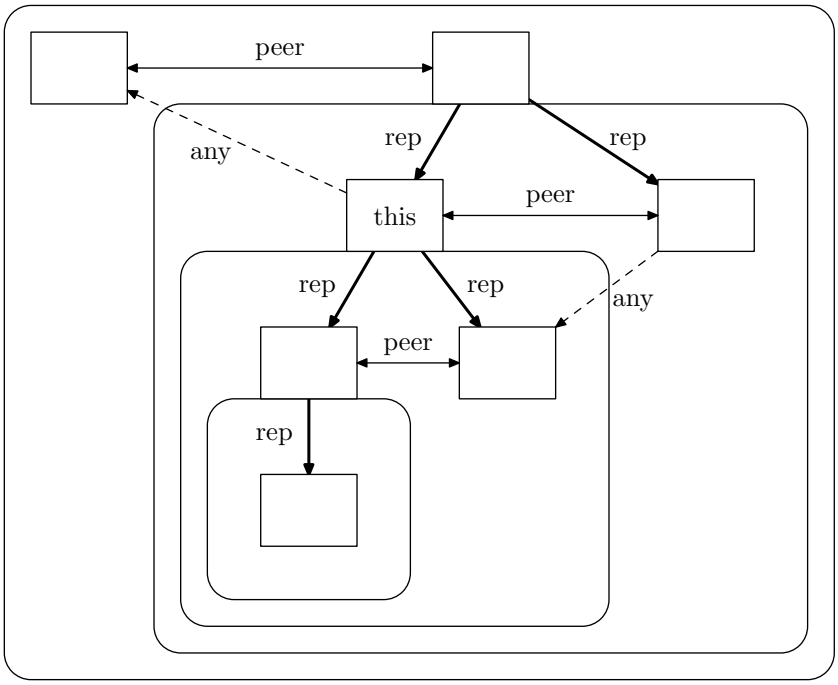

Figure 1: Ownership relations in an object structure.

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.

2 Directory Layout

The directory layout for the plugin sources was adopted from Apache Maven [10], even though the build system for the plugins employs Apache Ant [3]. Maven uses a standard directory layout which separates the implementation from the tests and generally follows best practices. Listing 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 5 for some of the most interesting properties.

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.

```
.
|-- 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 1: Directory layout of the plugin sources.

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 2 and Figure 3 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.

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 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

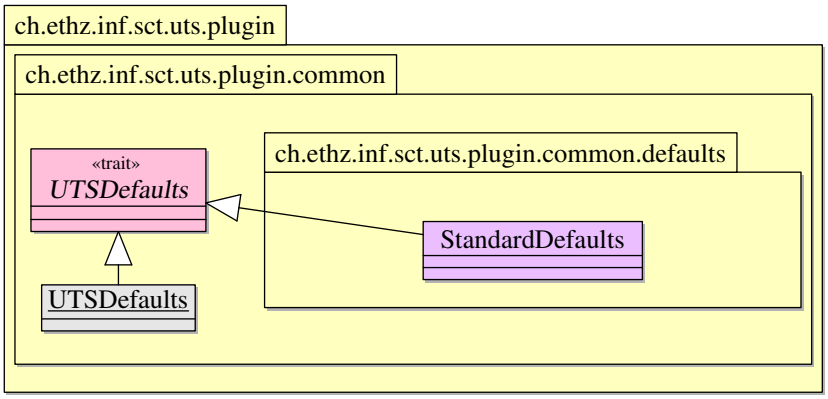

Figure 2: Defaults for the type checker.

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 either `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.

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 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 3 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 [9] 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.

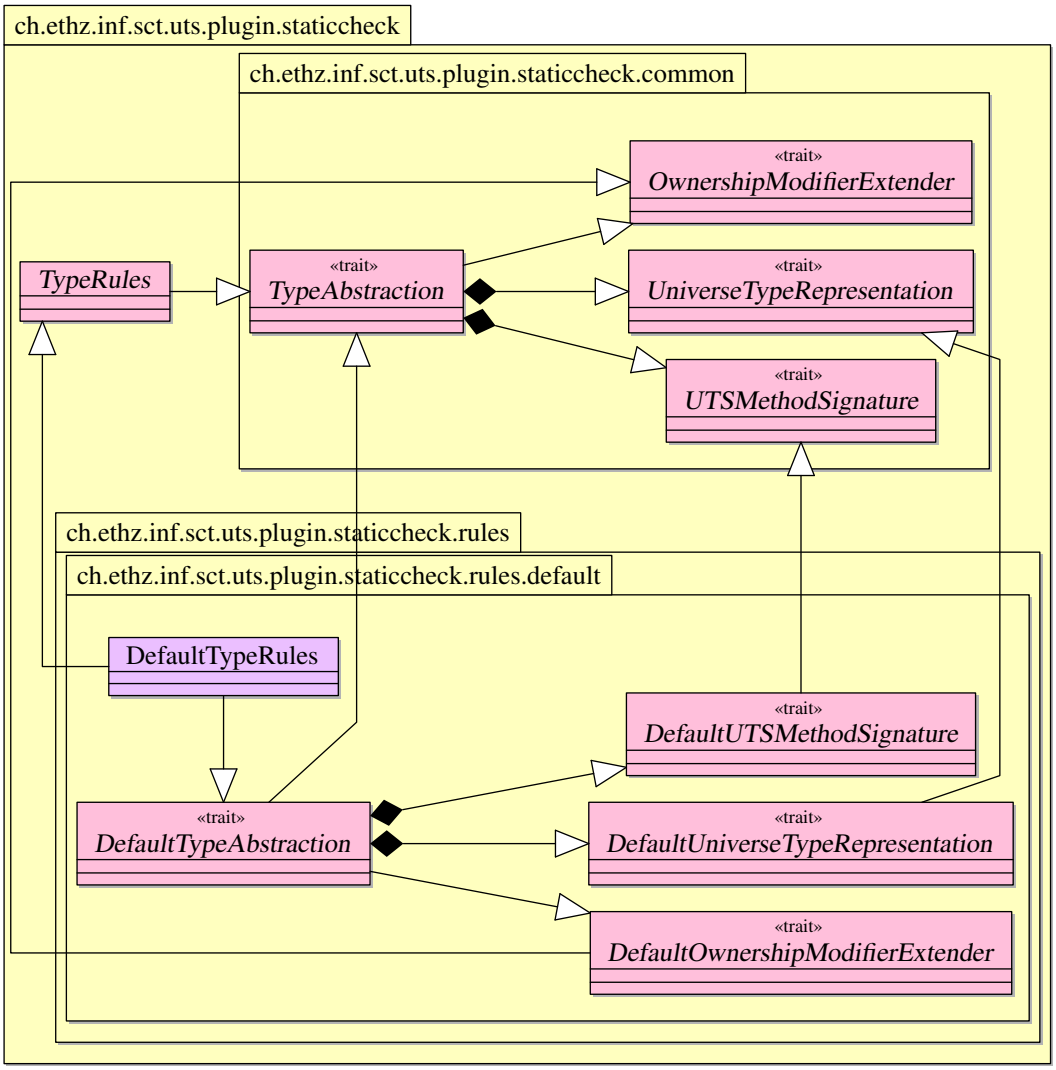

Figure 3: Implementation of type rules.

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 [5].

3.2.1 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.

3.2.2 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.

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 [6]. 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.

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`.

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 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.

6 Setting Up a Scala Bazaar

Scala Bazaars [7], 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 [1] (which was used in this guide) or Apache Tomcat [4] 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.

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 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.

6.2 Setup of the Bazaar

Listing 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.

```
<?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 2: `web.xml` for the `sbaz` servlet.

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

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

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

Listing 4: `keylessRequests` for setup.

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

Listing 5: `keylessRequests` for use.

```
<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 6: Bazaar descriptor for the `sbaz` tool.

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 4 which allows the retrieval of the package list and key management. For details about the contents of the file, see [8]. It can and should be replaced by a more restrictive configuration like the one in Listing 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.

6.3.1 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 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 `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

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

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.

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 6.3.1.

The key for the message pattern `<edit nameregex=".*"/>` created in Section 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.

References

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

[2] 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`.

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

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

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

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

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

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

[9] 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.

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