EffScript: Practical Effects for Scala

EffScript is a small domain-specific language for writing tailored effect disciplines for Scala. In addition to being customizable, the underlying effect system supports both effect polymorphism (as developed by Lukas Rytz in his PhD thesis) and gradual effect checking (following the theory of Bañados, Garcia and Tanter).

Scala Implementation

The implementation of the Practical Effect system is developed as a compiler plugin for the Scala programming language. The plugin is based on the plugin developed by Rytz *et al* and is composed of two sub plugins to implement bidirectional checking: the effect inference plugin, and the effect checking plugin. The effect inference plugin is a modification of the one developed by Rytz *et al*, extended with support for gradual effects and the customizable effect system. In relation to bidirectional checking, Scala has inference of effect, therefore there are cases where there is absence of effect annotations. The effect inference plugin is necessary to annotate function abstractions that do not have effect annotations. This information about effect inference is used by the effect checking plugin to check and adjust sets of effect privileges, also inserting runtime checks of effect wherever it may be necessary.

Getting Started Guide

The implementation

The code and examples can be downloaded here.

Alternatively an virtual box image can be downloaded here (http://www.pleiad.cl/paper_225_image.tar.gz) .

To run the examples, you need to have installed Scala (http://www.scala-lang.org/ (http://www.scala-lang.org/)), SBT (http://www.scala-sbt.org/ (http://www.scala-sbt.org/)) and JDK1.7 (http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html (http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html)).

In case we have Java 1.8 already installed, we need to change the `JAVA_HOME` variable so it use Java 1.7 instead. For example:

```
export JAVA_HOME=/Library/Java/JavaVirtualMachines/jdk1.7.0_79.jdk/Contents/HOME

PATH=$JAVA_HOME/bin:$PATH sbt
```

Compiling and packaging the inference and checking library

For the effect inference library:

```
cd efftp
sbt package
```

For the effect checking library:

```
cd gpes
sbt package
```

How to use the compiler plugins

Let us create new project that will use the effect system with a custom “simpleIO” discipline (see `eff-examples/simpleUI.eff` file). Note that we also provide a ready to use project in folder `playground-src`:

```
name: simpleIO

privileges:
        @simpleNoIO
        @simpleOutput
        @simpleInput

lattice:
    top: @simpleOutput @simpleInput
    bottom: @simpleNoIO

pointcuts:
        def views.html.dummy.apply() prod @simpleNoIO
        def views.html.foo.apply[T]() prod @simpleNoIO
        call scala.Predef.read*: T prod @simpleInput
        call fakePrint(T <: String) prod @simpleOutput
        call readM prod @simpleInput
        def scala.Predef.read*(V): T prod @simpleInput
```

We first create the folder for our new project which we call `playground`

```
mkdir playground
```

We need to link the compiler libraries to our new project by creating symbolic links of the jars inside the new project. Note that we could have also copied the libraries but modifications to the discipline implies performing a new copy after each compilation. To do this step we have two options:

1. Using the `copyCompilerLibs.sh` script\\At the root of the new project:

   ```
   chmod +x copyCompilerLibs.sh #make sure the file has execution permissions
   ./copyCompilerLibs.sh playground
   ```

   The `copyCompilerLibs.sh` script create two symbolic links to a folder lib at the root of the provided path.

2. Creating the symbolic links manually
   Alternatively we can create the symbolic links manually. The `copyCompilerLibs.sh` script is equivalent to (note that the path to the jar must be **absolute path**, using a relative path will not work properly):

   ```
   cd playground
   mkdir -p lib

   ln -s [Absolute-path]/efftp/target/scala-2.10/effects-plugin_2.10-0.1-SNAPSHOT.jar lib/infer.jar
   ln -s [Absolute-path]/gpes/target/scala-2.10/effects-checker-plugin_2.10-0.1-SNAPSHOT.jar lib/check.jar
   ```

Next we need to edit a `build.sbt` file to declare the project name, Scala version, libraries and compiler options.

The effect plugin works with Scala 2.10.3, which can be specified in every project by editing a `build.sbt` file at the root of the project.

Our `build.sbt` file looks like this:

```
name := "playground"

version := "0.1"

autoCompilerPlugins := true

scalaVersion := "2.10.4"

libraryDependencies ++= List( "org.scala-lang" % "scala-compiler" % "2.10.4" )


scalacOptions += "-Xplugin:lib/infer.jar"

scalacOptions += "-Xplugin:lib/check.jar"

scalacOptions += "-P:effects:domains:simpleIO"

scalacOptions += "-P:effects:unchecks:java.*:scala.(?!Function)*"
```

Next, let us create a file `helloGE.scala` to play with the effect system inside the folder “src/main/scala/”

Next, make sure `helloGE.scala` looks like this

```
import scala.annotation.effects._

object HelloGradualEffects{

  def main(args: Array[String]): Unit @simpleOutput = {
    println("hello gradual effects")
  }
}
```

Were the body of main is only allowed to produce `@simpleOutput`. We can test this by calling method “readInt” which produces `@simpleInput` according to the effscript file:

```
  def main(args: Array[String]): Unit @simpleOutput = {
    println("hello gradual effects")
    readInt()
  }
```

If we run the last code using sbt:

```
sbt
>compile
```

Sbt will try to compile it, but it will raise a static error:

```
[error]  found   : @simpleInput
[error]  required: @simpleOutput
[error]     readInt()
[error]            ^
[error] one error found
```

Now let us test gradual effects:

```
  def main(args: Array[String]): Unit @simpleOutput @pure= {
    def foo: Int @unknown = {
      readInt()
    }
    println("hello gradual effects")
    foo
  }
```

Running the code gives a runtime error:

```
>run
...
[error] (run-main-1) runtimePrivileges.RuntimePrivileges$EffectPrivilegeException: Not enough privileges:simpleInput()
[error]         at scala.this.Predef.readInt() (helloGE.scala:16)
runtimePrivileges.RuntimePrivileges$EffectPrivilegeException: Not enough privileges:simpleInput()
...
Caused by: runtimePrivileges.RuntimePrivileges$EffectPrivilegeException:
        The localized restriction to "simpleOutput()" did not set all the required privileges
        at foo (helloGE.scala:20)
        The outermost restriction to "simpleOutput()" did not set the required privileges
        at {
  def foo: Int @scala.annotation.effects.simpleNoIO @scala.annotation.effects.unknown = {
    scal... (helloGE.scala:13)
...
```

Step by Step Instructions

Architectural Constraints in Play

The effscript file to enforce the MVC pattern in Play with Effscript is defined in `eff-examples/DBAccess.eff`. To modify the discipline, just edit the `DBAccess.eff` file, and then run

```
cd effscript
./updateDBAccessDomain
```

The files to test the effect discipline are found in a play application located in folder `play-slick-advanced-effects`. To run the application just run:

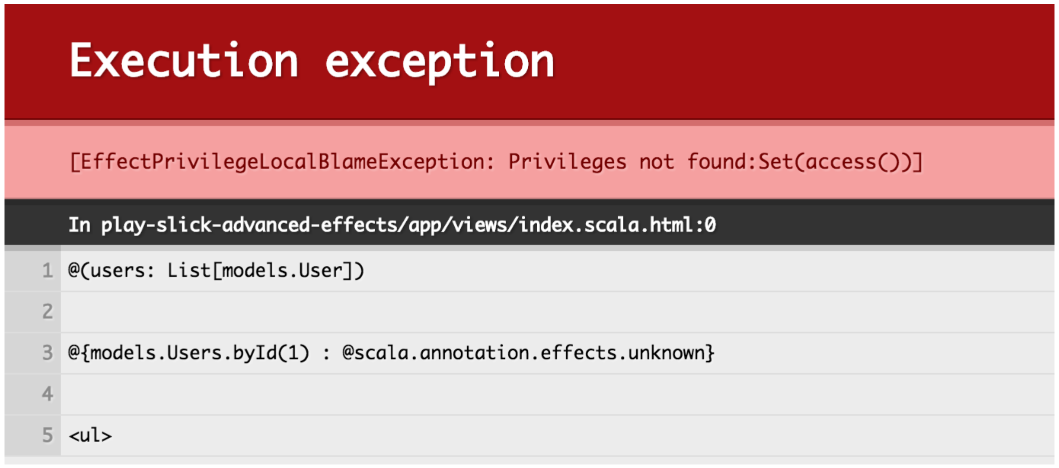
```
./copyCompilerLibs.sh play-slick-advanced-effects
cd play-slick-advanced-effects
./activator run
```

And just access http://localhost:9000 (http://localhost:9000) to test the effect discipline. The controller can be found in `app/controllers/EffectController.scala`. The models can be found in `app/models/` The template can be found in `app/views/index.scala.html`

To test the access to the database from the templates, just uncomment line 3 of file `app/views/index.scala.html` like this:

```
@{models.Users.byId(1) : @scala.annotation.effects.unknown}
```

And then reload the page:

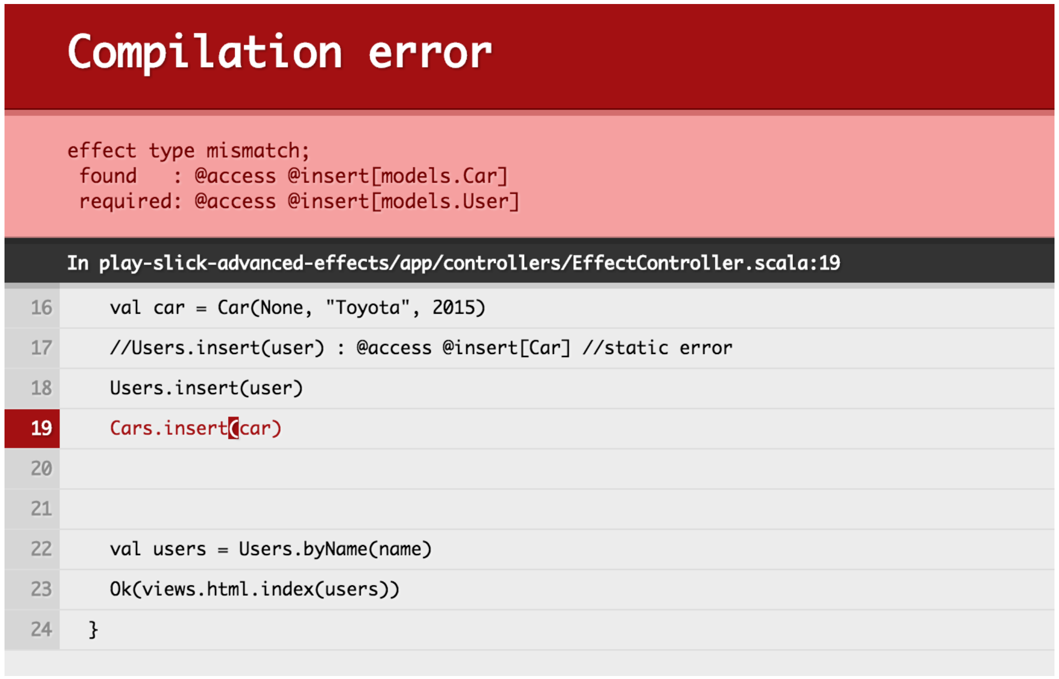


```
Execution exception

[EffectPrivilegeLocalBlameException: Privileges not found:Set(access())]

In play-slick-advanced-effects/app/views/index.scala.html:0

@(users: List[models.User])

@{models.Users.byId(1) : @scala.annotation.effects.unknown}

<ul>
```

To test that the EffectControllers only have access to insert `Users` just uncomment line 19 of file `app/controllers/EffectController.scala`:

```
Cars.insert(car)
```

And then reload the page to see how an static error is raised:



```
Compilation error

effect type mismatch;
 found   : @access @insert[models.Car]
 required: @access @insert[models.User]

In play-slick-advanced-effects/app/controllers/EffectController.scala:19

16     val car = Car(None, "Toyota", 2015)
17     //Users.insert(user) : @access @insert[Car] //static error
18     Users.insert(user)
19     Cars.insert(car)
20
21
22     val users = Users.byName(name)
23     Ok(views.html.index(users))
24   }
```

Modifying a discipline

Let us modify the simpleIO discipline defined in the previous section. Let us remember that the file can be found in `eff-examples/simpleIO.eff`.

To modify the discipline, just edit the `simpleIO.eff` file, and then run

```
cd effscript
./updateSimpleIODomain
```

Let us look at `updateSimpleIODomain`:

First it runs a script that uses effscript to generate the classes needed for the compiler plugins:

```
sbt "run ../eff-examples/simpleIO.eff"
```

then it copies the generated files into each of the compiler plugins, overriding the existing implementations with the new ones:

```
DIR=$( cd "$( dirname "${BASH_SOURCE[0]}" )" && pwd )

cp target/SimpleIO.scala $DIR/../gpes/src/main/scala/annotation/effects/SimpleIO.scala

cp target/SimpleIO.scala $DIR/../efftp/src/main/scala/scala/annotation/effects/SimpleIO.scala

cp target/SimpleIODomain\(check\).scala $DIR/../gpes/src/main/scala/simpleIO/SimpleIODomain.scala

cp target/SimpleIODomain\(infer\).scala $DIR/../efftp/src/main/scala/scala/tools/nsc/effects/simpleIO/SimpleIODomain.scala
```

After that step, we need to re-package both compiler plugins updating the `.jar` files.

Creating a new discipline

To create a new discipline we need to do extra steps. Let us suppose we want to add a new discipline called “newDisc”. Once we have created our `newDisc.eff` file we need to create the folders for this discipline in every compiler plugin project:

```
mkdir gpes/src/main/scala/newDisc
mkdir efftp/src/main/scala/scala/tools/nsc/effects/newDisc
```

Then create and run a batch just like the previous step to generate the files and copy them into each compiler plugin project.

Next, we need to add this discipline into each of the compiler plugin projects:

For “gpes” we need to edit `src/main/scala/EffectsCheckerPlugin.scala` file, and add a new case inside `mkDomains` function:

```
        ...
        case "simpleTPIO" :: xs =>
         new simpletpio.SimpleTPIODomain {
           val global: EffectChecker.this.global.type = EffectChecker.this.global
           override val uncheckedFunctions = settings.unchecks
         } :: mkDomains(xs)
        case "newDisc" :: xs =>
         new newdisc.NewDiscDomain {
           val global: EffectChecker.this.global.type = EffectChecker.this.global
           override val uncheckedFunctions = settings.unchecks
         } :: mkDomains(xs)
        case x :: xs => mkDomains(xs)
```

For “efftp” we need to edit `src/main/scala/tools/nsc/effects/EffectsPlugin.scala` file, and add a new case inside `mkDomains` function:

```
     ...
    case "simpleTPIO" :: xs =>
      new simpletpio.SimpleTPIODomain {
          val global: EffectsPlugin.this.global.type = EffectsPlugin.this.global
          override val uncheckedFunctions = efftpSettings.unchecks
        } :: mkDomains(xs)
    case "newDisc" :: xs =>
      new newdisc.NewDiscDomain {
          val global: EffectsPlugin.this.global.type = EffectsPlugin.this.global
          override val uncheckedFunctions = efftpSettings.unchecks
        } :: mkDomains(xs)
    case x :: xs =>
      global.abort(s"Unknown effect domain: $x")
```

Next, just re package each of the compiler plugin projects and update libraries of every project accordingly (unless symbolic links are being used).

Running Benchmarks

Benchmarks are inside `benchmarks` folder.

Using Scala 2.10.x

For benchmarks, it is better to run them not using sbt as it is faster. To run the benchmarks using the `scala` command you need to have the Scala version 2.10.x. For example, to run a benchmark using the scala command we can use the `runExperiment` script file:

```
#!/bin/bash
/usr/local/Cellar/scala210/2.10.4/bin/scala -cp CollsSimple/lib/check.jar:/usr/local/Cellar/scala210/2.10.4/libexec/lib/scala-reflect.jar $1 $2
```

Notice how the scala command and the reflection library correspond to version 2.10.4. We can run a “benchmark run” like this (edit this file to indicate the appropriate path for Scala binaries and libraries):

```
./runExperiment [jar] [version]
```

Where [version] can be:

- 0 ⇒ 95 dynamic checks and 67 context adjustements
- 1 ⇒ 35 dynamic checks and 67 context adjustements
- 2 ⇒ 35 dynamic checks and 67 context adjustements
- 3 ⇒ fully annotated (do not produce runtime effect checks)

The [jar] can be:

- `CollsNoeffs/target/scala-2.10/collsnoeffs_2.10-0.1.jar` ⇒ without the effect system
- `CollsSimple/target/scala-2.10/collssimple_2.10-0.1.jar` ⇒ default effect system.
- `CollsSimpleBits/target/scala-2.10/collssimple_2.10-0.1.jar` ⇒ default effect system using bit vectors.
- `CollsSimpleSE/target/scala-2.10/collssimplese_2.10-0.1.jar` ⇒ scenario with subeffecting.
- `CollsSimpleTP/target/scala-2.10/collssimpletp_2.10-0.1.jar` ⇒ scenario with subeffecting and type parameters.

For example, to run the default effect system with the version with most dynamic checks:

```
./runExperiment CollsSimple/target/scala-2.10/collssimple_2.10-0.1.jar 0
```

The program returns the number of second of the benchmark.

Using SBT

In case we do not have Scala 2.10 we can always run each benchmark by entering to the folder of the experiment. The folders are the following:

- `CollsNoeffs` ⇒ without the effect system
- `CollsSimple` ⇒ default effect system.
- `CollsSimpleBits` ⇒ default effect system using bit vectors.
- `CollsSimpleSE` ⇒ scenario with subeffecting.
- `CollsSimpleTP` ⇒ scenario with subeffecting and type parameters.

For instance:

```
cd CollsSimple
```

The versions of each experiment are the following:

- 0 ⇒ 95 dynamic checks and 67 context adjustements
- 1 ⇒ 35 dynamic checks and 67 context adjustements
- 2 ⇒ 35 dynamic checks and 67 context adjustements
- 3 ⇒ fully annotated (do not produce runtime effect checks)

For instance, to run the version 2 of the experiment, we run:

```
sbt "run 2"
```