Parsing with First-Class Derivatives

Artifact Evaluation Overview for Paper #138

Double-blind submission

Abstract

The conclusions of OOPSLA 2016 submission #138 are supported by the artifact we submit for evaluation. As required, this overview consists of a getting-started guide and step-by-step instructions for how to evaluate our artifact.

1. Getting Started

This section describes how to get started interacting with our artifact. Our artifact is a directory structure full of Scala code. We use sbt, a Scala build tool, to compile and run all of the Scala code.

1.1 Contents of `138.zip`

The `138.zip` file we submit for artifact evaluation contains the following files and directories:

`artifact/`	Scala source code of our artifact
`configs/`	Auxiliary files for the virtual machine setup
`paper.pdf`	our paper as submitted
`overview.pdf`	this overview
`Vagrantfile`	Configuration file that allows automated setup of a virtual machine

1.2 Setting up Your Environment

As you may note from the contents listing above, there are two ways to run and interact with our artifact. Please note, that *both ways require connection to the internet*.

- **Run the Scala code inside a virtual machine**. We prepared a `Vagrant` config file that, when executed, sets up a virtual machine for you and downloads and installs the necessary dependencies inside the virtual machine. So all you need is *Virtual Box*, *Vagrant* and internet connect. See Section 1.3 for instructions.
- **Run the Scala code on your computer**. This probably is the easiest variant, if you already have some version of Scala and sbt installed. When compiling our artifact with sbt, the required Scala version will automatically be installed. So all you need is a text editor, a Java runtime environment, and internet connection. See Section 1.5 for instructions.

Choose how you want to interact with our artifact and follow the setup instructions in the corresponding subsection.

1.3 Setup on Virtual Machine

To ensure a reproducable setup of the necessary environment to interact with our artifact, we provide a Vagrant[1] script that automatically sets up the virtual machine and downloads all necessary dependencies.

To be able to use the virtual machine, you first need to install the necessary virtualization player, as well as Vagrant on your host machine. We tested the Vagrant script and the resulting virtual machine image with Virtual Box 5.0.16 and Vagrant 1.7.4 on Mac OS X Yosemite, as well as Virtual Box 4.3.26 and Vagrant 1.7.4 on Windows 7 Enterprise.

If you have the right version of Virtual Box and Vagrant installed, open a terminal on your host machine and navigate to the folder that contains the `Vagrantfile`. Enter

```
$ vagrant up --provision
```

to initialize the virtual machine. This will trigger the download of the base image (an Ubuntu system) followed by the download and configuration of several dependencies inside the virtual machine. This can take a couple of minutes and depends on your internet connection.

If everything is working fine and the initialization process finished successfully, enter

```
$ vagrant ssh
```

to access a terminal in the virtual machine. You will find the following directory structure within (`/home/vagrant/`):

`~/artifact`	source code of our artifact, shared with the equally named folder on your host machine.
`~/bin`	sbt binaries used to compile and run our artifact.
`~/configs`	Auxiliary files for the virtual machine setup, shared with the equally named folder on your host machine.

[1] `https://www.vagrantup.com/`

1.4 Interacting with SBT

To start interacting with our artifact, change the working directory to `~/artifact` and then run `sbt`. You are now faced with an sbt prompt similar to Figure 1. Try entering `sbt-version`. sbt should answer with `[info] 0.13.8`.

Since the folder `artifact/` is shared between the VM and your host machine, you can use your favorite text-editor or IDE on your host machine to inspect and edit the sources of our artifact. Compilation and execution of our artifact is then performed within the VM using `sbt` (also see Section 1.4).

Note to Windows Users. `sbt` will store all compiled class files and temporary files in the folder `artifact/target` which is shared between the VM and your host machine. You do not need to interact with those files. However, `sbt` also attempts to create symlinks in this folder which fails on Windows systems. As a workaround, please enter the following command after starting `sbt` to change the target-path to a folder that is not shared between the VM and your host machine:

```
$ sbt
> set target:=file("/home/vagrant/target/")
[info] Defining *:target
...
>
```

We apologize for this inconvenience.

1.5 Setup on Your Machine

Make sure you have `sbt`[2] installed on your system. Then unzip `138.zip` and launch sbt from the `artifact` directory.

When you run `sbt` for the first time, it will download various components, possibly including the correct version of sbt itself and the Scala compiler.

Eventually, the terminal should display a prompt for entering sbt commands. At that point, the terminal should look a bit like Figure 1. Try entering `sbt-version`. sbt should answer with `[info] 0.13.8`. If this works, you are set up for continuing with Section 1.4.

At the sbt prompt, you can enter various commands. Most important are:

- `exit` to exit sbt.
- `compile` to compile all Scala files in the artifact.
- `console` to open a Scala REPL with the project in scope. You can enter simple expressions like `3 + 5` to evaluate them. Most examples mentioned in the step-by-step guide or related Scala files can simply be entered in the console. For instance try to run the method `paper.section_3_2.number.parse("42")`. It should return `List(42)`. Use Ctrl-D to exit the REPL and get back to sbt.
- `test` to execute our prepared test suite and additional tests you write to evaluate the artifact.

We recommend that you run all the tests we prepared by entering `test` in the sbt-console. Running the tests might fail, if there is not enough free memory.

Note that sbt will figure out dependencies and might download and build more code for you. sbt is also clever about not recompiling files if that is not necessary.

1.6 Project Layout

When working with sbt the user has to be aware of the project layout which is imposed by convention. When searching for a Scala file there might be four possible locations to look at:

- `/project` contains the description of the build system and all scala code that is compiled and executed *before* the other source files are being compiled. You can probably safely ignore this folder.
- `/src/main/scala` contains implementation of our parser combinator library, as well as the implementation of some derived combinators.
- `/src/test/scala` contains the test suite we prepared to develop and evaluate our artifact.

  As part of the step-by-step guide the subfolder `paper` contains annotated scala code.

2. Step-by-Step Evaluation

We submit three different artifacts that we believe support our claims made in the paper.

1. implementation of a parser combinator library based on parsing with derivatives that also supports first-class derivatives.
2. implementation of standard and novel derived combinators.
3. several small case studies.

All of the artifacts consist of Scala source code, and we believe that working with this code is the best way to evaluate it. We therefore put most of the information for the artifact reviewers as comments in source files. In this way, the information can be right next to working examples that the reviewers can change to experiment with our artifact. Figure 2 gives a listing of the files in the `src/main/scala/examples` directory. `Section3.scala` can serve as an entry point to evaluate the artifact.

The implementation of our library itself can be found in the `src/main/scala/library/` director. The listing in Figure 3 should provide some overview on how the implementation is structured.

[2] `http://www.scala-sbt.org`

We briefly want to repeat the claims of our paper and point to the respective sections in the code that we believe support our claim.

"We implemented a parser combinator library that features first-class derivatives." We believe this claim is mostly but not exclusively backed by `DerivativeParsers.scala` - see also `Section4.scala`).

"Parsers defined with fcd are modular and facilitate reuse." This artifact includes various examples of parser combinators for indentation (`Section3.scala` and `PythonParsers.scala`), mixed content and ASCII tables (`Section5.scala`).
`Section5.scala` also illustrates how multiple such combinators can be combined to assemble a parser for complex languages from components.

"With our library, the input stream of a parser is not necessarily a continuous substream of the original (physical) input stream." Examples involving 'collect' illustrate this (`Section3.scala`) but most of the parser combinators in this artifact build on this fact.

We do not claim that our library / example parser implementations are production ready. It might be possible that reviewers encounter "Out of memory" exceptions. The goal of our approach is to show how first class derivatives can improve modularity of parser implementations. We hope to solve the performance issues in future work.

```
[info] Loading project definition from .../138/artifact/project
[info] Updating {file:.../138/artifact/project/}artifact-build...
[info] Done updating.
[info] Set current project to first-class-derivatives (in build file:.../138/artifact/)
>
```

Figure 1. After downloading various components and compiling the project definition, sbt shows a prompt.

`paper/Section3.scala`	code examples from Section 3 of the paper. Also contains additional introductory notes.
`paper/Section4.scala`	code examples from Section 4 of the paper. Also contains notes on the relation between the paper and the artifact implementation.
`paper/Section5.scala`	code examples from Section 5 of the paper. Also contains some additional examples.
`PythonParsers.scala`	a case study (not reported in the paper) of a Python parser supporting more advanced concepts of Python indentation. The individual indentation-requirements from the Python specification are implemented as separate combinators which are then composed.

Figure 2. Contents of the `src/main/scala/examples` directory.

`Parser.scala`	The interface of our parser combinator library as introduced in Section 3. This file corresponds to Figure 1a.
`Syntax.scala`	Contains all the tricks to support a Scala syntax close to the paper. This file corresponds to Figure 1b in Section 3.
`DerivedOps.scala`	Derived combinators such as those in Figure 1c, Figure 5a and Figure 9b.
`CharSyntax.scala`	Derived combinators and parsers specific to a token type `Char`.
`DerivativeParsers.scala`	Our implementation of derivative based parsing, implementing the interface of `Parser.scala`. This file corresponds to Figure 6 in the paper.
`Printable.scala`	Helper functions to print graphical representations of parsers.
`Attributed.scala`	Implementation of attributes that are defined as fixed points. Slightly adapted implementation from Matt Might.

Figure 3. Contents of the `src/main/scala/library` directory.