Generating a fluent API with syntax checking from an LR grammar (Artifact)

Tetsuro Yamazaki Tomoki Nakamaru Kazuhiro Ichikawa Shigeru Chiba
The University of Tokyo

July 8, 2019

1 Introduction

We have implemented a fluent API generator, named `typelevelLR`, for Scala, Haskell, and C++. It generates a library skeleton, which uses the host-language type checker to detect an invalid chain of method calls to the library. The valid method-call chains are specified by an LR grammar. This document describes how to build `typelevelLR` and run our experiments.

All the source code is available from our github repository:

`https://github.com/csg-tokyo/typelevelLR`

The hash of the commit for that source code is:

`d36b70a56c5b83a8a2b7006ca44d23f41f9a0843`

2 How to build and install

First we build `typelevelLR`.

1. Install stack (`https://docs.haskellstack.org/`), a build tool for Haskell.

   ```
   $ curl -sSL https://get.haskellstack.org/ | sh
   ```

2. Append `$HOME/.local/bin` to `PATH`

   ```
   $ export PATH=$PATH:$HOME/.local/bin
   ```

   We used version 1.6.3.

3. Clone our github repository

   ```
   $ git clone -b OOPSLA2019-artifact --single-branch
         https://github.com/csg-tokyo/typelevelLR
   ```

4. Build and install `typelevelLR`.

   ```
   $ cd typelevelLR
   $ stack install
   ```

   Stack will copy the compiled binary to `$HOME/.local/bin`.

5. Try it

```
$ typelevelLR --help
Usage: typelevelLR [OPTION...]
  Generate a fluent API library skeleton from an LR Grammar

Options:
           --hs, --haskell  generate Haskell library
           --cpp            generate C++ library
           --scala          generate Scala library
  -d PATH  --dest=PATH      destination directory path
  -s PATH  --source=PATH    source directory path
  -?       --help           display this help and exit
  -v[n]    --verbose[=n]    set verbosity level
```

3 Generate a library skeleton for Haskell

1. Prepare a syntax file

   ```
   $ vi hello.syntax
   ```

   ```
   syntax helloDSL (Start) {
     SimpleHello   : Start -> "hello"
     HelloWithName : Start -> "hello" "name(String)"
   }
   ```

   You can also copy `PATH-TO-typelevelLR/examples/haskell/hello/hello.syntax` into your working directory.

2. Generate a library skeleton

   ```
   $ typelevelLR --haskell
   $ ls HelloDSL.hs
   HelloDSL.hs
   ```

3. Use the generated library skeleton

   ```
   $ vi MyApp.hs
   ```

   ```
   import HelloDSL

   main :: IO ()
   main = print $ begin |> hello |> name "ymzk" |> end
   ```

   ```
   $ stack runghc MyApp.hs
   HelloWithName "ymzk"
   ```

   You can also copy `PATH-TO-typelevelLR/examples/haskell/hello/MyApp1.hs` into your working directory and run it.

4. Define semantics

   ```
   $ vi HelloDSLSemantics.hs
   ```

   ```
   module HelloDSLSemantics where

   import HelloDSL

   runHelloDSL :: Start -> IO ()
   runHelloDSL SimpleHello          = putStrLn "Hello."
   runHelloDSL (HelloWithName name) =
       putStrLn ("Hello, " ++ name ++ "!!")
   ```

You can also copy `PATH-TO-typelevelLR/examples/haskell/hello/HelloDSLSemantics.hs` into your working directory.

5. Run it

```
$ vi MyApp.hs

import HelloDSL
import HelloDSLSemantics

main :: IO ()
main = runHelloDSL $ begin |> hello |> name "ymzk" |> end

$ stack runghc MyApp.hs
Hello, ymzk!!
```

You can also copy `PATH-TO-typelevelLR/examples/haskell/hello/MyApp2.hs` into your working directory and run it.

4 Generate a library skeleton for Scala

1. Prepare a syntax file

```
$ vi hello.syntax

syntax helloDSL (Start) {
  SimpleHello   : Start -> "hello"
  HelloWithName : Start -> "hello" "name(String)"
}
```

You can also copy `PATH-TO-typelevelLR/examples/scala/hello/hello.syntax` into your working directory.

2. Generate a library skeleton

```
$ typelevelLR --scala
$ ls helloDSL.scala
helloDSL.scala
```

3. Use generated library skeleton

```
$ vi myApp.scala

object myApp {
  import helloDSL._
  def main(args: Array[String]) = {
    val parseTree: Start = begin().hello().name("ymzk").end()
    println(parseTree)
  }
}

$ scalac -sourcepath . myApp.scala && scala myApp
HelloWithName(ymzk)
```

You can also copy `PATH-TO-typelevelLR/examples/scala/hello/myApp1.scala` into your working directory and run it.

4. Define semantics

```
$ vi helloDSLSemantics.scala

object helloDSLSemantics {
  import helloDSL._
  def runHelloDSL(parseTree: Start) = parseTree match {
    case SimpleHello() => println("Hello.")
    case HelloWithName(name) => println(s"Hello, $name!!")
  }
}
```

You can also copy `PATH-TO-typelevelLR/examples/scala/hello/helloDSLSemantics.scala` into your working directory.

5. Run it

```
$ vi myApp.scala

object myApp {
  import helloDSL._
  import helloDSLSemantics._
  def main(args: Array[String]) = {
    val parseTree: Start = helloDSL.begin().hello().name("ymzk").end()
    runHelloDSL(parseTree)
  }
}

$ scalac -sourcepath . myApp.scala && scala myApp
Hello, ymzk!!
```

You can also copy `PATH-TO-typelevelLR/examples/scala/hello/myApp2.scala` into your working directory and run it.

5 Generate a library skeleton for C++

1. Prepare a syntax file

```
$ vi hello.syntax

syntax helloDSL (Start) {
  SimpleHello   : Start -> "hello"
  HelloWithName : Start -> "hello" "name(std::string)"
}
```

You can also copy `PATH-TO-typelevelLR/examples/c++/hello/hello.syntax` into your working directory.

2. Generate a library skeleton

```
$ typelevelLR --cpp
$ ls helloDSL.*
helloDSL.cpp      helloDSL.hpp      helloDSL.hpp.impl
```

3. Use the generated library skeleton

```
$ vi myapp.cpp
```

```
#include <iostream>
#include <memory>
#include "helloDSL.hpp"
using namespace helloDSL;

int main() {
  std::shared_ptr<Start> parseTree = begin()->hello()->name("ymzk")->end();
  std::cout << *parseTree << std::endl;
  return 0;
}
```

```
$ g++ -std=c++17 -c helloDSL.cpp
$ g++ -std=c++17 -c myapp.cpp
$ g++ -o myapp helloDSL.o myapp.o
$ ./myapp
HelloWithName(ymzk)
```

You can also copy `PATH-TO-typelevelLR/examples/c++/hello/myapp1.cpp` into your working directory and run it.

4. Define semantics

```
$ vi helloDSLSemantics.hpp
```

```
#include <iostream>
#include <memory>
#include "helloDSL.hpp"
using namespace helloDSL;

class HelloDSLSemantics : public Start::ConstVisitor {
  void visitSimpleHello(const SimpleHello& host) {
    std::cout << "Hello." << std::endl;
  }
  void visitHelloWithName(const HelloWithName& host) {
    std::cout << "Hello, " << std::get<0>(host) << "!!" << std::endl;
  }
};

void runHelloDSL(const std::shared_ptr<Start>& parseTree) {
  static HelloDSLSemantics semantics;
  parseTree->accept(semantics);
}
```

You can also copy `PATH-TO-typelevelLR/examples/c++/hello/helloDSLSemantics.hpp` into your working directory.

5. Run it

```
$ vi myapp.cpp
```

```
#include <iostream>
#include <memory>
#include "helloDSL.hpp"
#include "helloDSLSemantics.hpp"
using namespace helloDSL;

int main() {
  std::shared_ptr<Start> parseTree = begin()->hello()->name("ymzk")->end();
```

```
  runHelloDSL(parseTree);
  return 0;
}

$ g++ -std=c++17 -c helloDSL.cpp
$ g++ -std=c++17 -c myapp.cpp
$ g++ -o myapp helloDSL.o myapp.o
$ ./myapp
Hello, ymzk!!
```

You can also copy `PATH-TO-typelevelLR/examples/c++/hello/myapp2.cpp` into your working directory.

6 Reproduction of our experiment

We wrote a Ruby script that reproduces our experiment shown in our paper. The script file is `PATH-TO-typelevelLR/experiment/experiment.rb` and it runs on Ruby 2.4.1. Our experiment shows the following claims:

1. Our tool, `typelevelLR`, is able to generate library-skeletons from grammar definitions, and
2. It does not take exponential time when compiling a method chain using the generated library-skeleton.

Our Ruby script performs the following four steps:

1. Reads the grammar file of our DOT-like language from the current directory,
2. Invokes `typelevelLR` to generate the library-skeleton,
3. Generates a number of source files including a method-call chain to the library, and
4. Measure their compilation time.

The DOT language is a domain-specific language for drawing a graph. So the method-call chain generated at the step 4 calls the library methods to draw a graph.

Before running this script, we have to move the current directory to the directory where the grammar file is found. We used two versions of the grammar file:

- Right-recursive version: `PATH-TO-typelevelLR/examples/LANGUAGE/dot-r/dot-r.syntax`
- Left-recursive version: `PATH-TO-typelevelLR/examples/LANGUAGE/dot-l/dot-l.syntax`

Here, `LANGUAGE` is either `haskell`, `scala`, or `c++`. So to run the script for the right-recursive version of the grammar and Haskell, we type as follows:

```
$ cd PATH-TO-typelevelLR/examples/haskell/dot-r
$ ruby PATH-TO-typelevelLR/experiment/experiment.rb --haskell -n 5
$ ruby PATH-TO-typelevelLR/experiment/experiment.rb --haskell > experiment.out
```

The second command will take a minute but the third one takes a day.

The Ruby script takes an option, `--haskell`, `--scala`, or `--cpp`, to choose the target langauge among Haskell, Scala, or C++, respectively. The script may take an option `-n`, which specifies the maximum number of nodes in the graph drawn by the method-call chian at the step 4. As a larger number is passed with `-n`, a longer method chain is generated. The default value for `-n` is 100. The Ruby script generates intermediate files under the directory `WORKSPACE_***`, where *** is the language name. If you use ghc 8.* or later one, our script with `--haskell` option reports

warnings like: "`-fcontext-stack=2000` is deprecated: use `-freduction-depth=2000` instead." If you don't want to see this, use an older version of ghc (e.g. ghc 7.10.3).

The output file is in the CSV format. It has five columns, n, d, e, *mean*, and *variance*. n is the number of the graph nodes. d represents the complexity of the graph drawn by the method-call chains. In our experiment, d is always 1. The generated graph contains $n \times d$ edges. e represents the length of the generated method chain. The script measures the compilation time 20 times for each pair of n and d and the script reports the mean and variance.

We ran our experiments on a machine with Intel CoreTM i7-4770S, 16 GB memory, and Ubuntu-16.04.5 LTS. We also used the Scala compiler `scalac 2.11.6` with the option `-J-Xss100m`, the Glasgow Haskell compiler `ghc 7.10.3` with `-O2 -fcontext-stack=5000`, and the GNU C++ compiler `g++ 5.4.0 20160609` with `-O2 -std=c++17`. The Scala compiler was run on the JVM 1.8.0_101.