MPRI course 2-4
“Programmation fonctionnelle et systèmes de types”
Programming project

Yann Régis-Gianas

2018–2019

1 Summary

The purpose of this programming project is to implement a compiler from the simply-typed λ-calculus to a categorical language. This technique is applied to automatically differentiate first-order numerical programs. This project is based on the following Conal Elliott’s papers:

- "The Simple Essence of Automatic Differentiation"[3]
- "Compiling to categories"[2]

We strongly advise you to read these papers before you start the project.

2 Required software

To use the sources that we provide, you need OCaml and Menhir. Any reasonably recent version should do. You also need the OCaml package `pprint`. If you have installed OCaml via `opam`, issue the following command:

```
opam install menhir pprint
```

3 Overview of the provided sources

Many components of the project are already provided, including: the definitions of the syntax of terms for the source and target language; a lexer and parser; a pretty-printer to OCaml programs. Most of the tasks consist in completing a specific module. In the `src/` directory, you will find the following files:

joujou.ml This is the “driver” which orchestrates the application of the compiler passes.

options.ml These are the program parameters and their declarations as command-line options.

source.ml This module contains the definition of the abstract syntax tree of the source language as well as several utilities to deal with these trees. The source language is a simply typed λ-calculus with tuples, floating-point literals and numerical primitives.

position.ml This module provides a type for source code locations used in error messages.

IO.ml, lexer.mll, parser.mly These modules provide a parser for the source language.

target.ml This module contains the definition of the target language, a language made of categorical combinators. There are no binder in that language.

oCamlGenerator.ml This module turns a program of the target language into an OCaml program parameterized by a category.

category.ml Here are the categories to instanciate the OCaml programs generated by the compiler.
This file will be completed during Task 1.

typechecker.ml This module implements a typechecker for the source language.
This file will be completed during Task 2.

compiler.ml This module implements a compiler for the source language to the target language.
This file will be completed during Task 3.

simplifier.ml This is a simplification procedure for programs written in the target language.
This file will be completed during Task 4.

tiny_mlp.j This file implements a very small Multi Layer Perceptron in the source language.
This file will be completed during Task 5.

tiny_mlp_test.ml This module trains the Multi Layer Perceptron to have it learn a simple function.
This file will be completed during Task 5.

Testing For each task, there is a corresponding testsuite in the `src/tests/` directory. Running `make` from `src/tests` executes them all.

Advice We strongly recommend that you regularly take checkpoints (that is, snapshots of your work) so that you can later easily roll back to a previous consistent state in case you run into an unforeseen problem. Using a versioning tool such as `git` is highly recommended.

4 Task description

Task 1 As a first task, you must complete `category.ml` using the definitions found in Conal Eliott's papers. This exercise is merely about translating Haskell code into OCaml code. The difficulty here is to make explicit what is hidden by Haskell overloading, that is, the actual type of operators. Looks at the numerous comments in the file, they should help you.

Task 2 For this second task, you must complete `typechecker.ml`. You must implement a type checker for the source language. You must also define a program transformation to eta-expanse all toplevel definitions since this is a precondition of the compiler. The main difficulty here is the fact that the testsuite for this task is intentionally small. It is therefore suggested to extend this testsuite with your own tests.

Task 3 This third task requires the completion of `compiler.ml`. This compiler translates a source program into a categorical language using a translation described in the paper "Compiling to categories"[2]. There are two main difficulties here: (i) the translation is not formally defined in the paper (hence, you will have to figure out what is its actual precise definition before digging into the implementation) ; (ii) the compiler must construct witnesses for the `ok` constraints required by the categorical combinators.

Task 4 As a fourth task, you will complete `simplifier.ml`. This module implements some rewriting rules over compiled programs to try to remove `curry` and `apply` combinators. This is needed to actually differentiate programs because the category of differentiable functions is not closed. There are two main difficulties in this task: (i) to exhibit the relevant laws to remove `curry` and `apply` from compiled programs ; (ii) to correctly implement the simplification procedure in particular to take the associativity of composition into account.

Task 5 As a final task, you will apply your differentiator to train a Multi-Layer Perceptron (MLP). To that end, you will complete `tiny_mlp.j` which defines a very small MLP from `float * float` to `float`. Then, you will complete `tiny_mlp_test.ml` to implement the learning procedure for this MLP. The idea is minimize the error with a gradient descent based on the automatically generated derivative[1].

Optional tasks If you wish to go further and receive extra credit, there are a number of things that you might do. Here are some suggestions. This list is not sorted and not limiting. Not all suggestions are easy! Think before attacking an ambitious extension.

- You can try to turn the type checker into a type inference engine to remove all the type annotations from programs.
- You can implement the `Dual` category defined in the paper[3].
- As you will discover during Task 5, the compiled programs generated by our compiler and differentiator are extremely slow. By picking ideas from Conal Elliott[3], you can try to optimize vector representations, variable representation, and more generally the shape of the generated programs. You can also try to "defunctorize" these programs to reduce the execution cost of modular abstractions.

In each case, please write **a textual explanation** of what you did, how you did it, and where to look for it in your code. Also, propose **test files** that illustrate what you did.

5 Evaluation

Assignments will be evaluated by a combination of:

- **Testing**. Your project will be tested with the testsuite that we provide (make sure that "`make -C tests`" succeeds!) and with additional tests.
- **Reading**. We will browse through your source code and evaluate its correctness and elegance.

6 What to turn in

When you are done, please e-mail to François Pottier and Pierre-Évariste Dagand and Yann Régis-Gianas and Didier Rémy a `.tar.gz` archive containing:

- All your source files.
- If you implemented "extra credit" features, a `README.md` file (written in French or English) describing these additional features, how you implemented them, and where we should look in the source code to see how they are implemented.

7 Deadline

Please turn in your assignment on or before **Friday, March 1, 2019**.

References

[1] Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of Machine Learning Research*, 18:153:1–153:43, 2017.

[2] Conal Elliott. Compiling to categories. *PACMPL*, 1(ICFP):27:1–27:27, 2017.

[3] Conal Elliott. The simple essence of automatic differentiation. *PACMPL*, 2(ICFP):70:1–70:29, 2018.