Minicaml, a purely functional, didactical programming language with an interactive REPL. WORK IN PROGRESS DRAFT

Alessandro Cheli
Course taught by Prof. Gianluigi Ferrari
and Prof. Francesca Levi

December 25, 2019

Abstract

minicaml is a small, purely functional interpreted programming language with a didactical purpose. It is based on the Prof. Gianluigi Ferrari and Prof. Francesca Levi's minicaml, an evaluation example to show students attending the Programming 2 course at the University of Pisa how interpreters work. It is an interpreted language featuring a Caml-like syntax, static (lexical scoping), interchangeable eager and lazy evaluation and a didactical REPL that shows each AST expression and each evaluation step.

1 REPL and command line interface

1.1 Installation

minicaml is available in the opam 2.0 repository. (`https://opam.ocaml.org/`). The easiest way to install minicaml is with the OCaml package manager **opam**. To do so, please check that you have a version of opam $\geq$ 2.0.0 and run:

```
opam install minicaml
```

Alternatively, **minicaml** can be installed from source by downloading the source code git repository and building it manually. **minicaml** has been tested only on Linux and macOS systems. It has not been tested yet on Windows and BSD derived systems.

```
# download the source code
git clone https://github.com/0x0f0f0f/minicaml
# cd into the source code directory
cd minicaml
# install dependencies
opam install ANSITerminal dune ppx_deriving menhir cmdliner
# compile
make
# execute
make run
# install
make install
```

2 Lexer

3 Parser

4 AST Optimization

Before being evaluated, AST expressions are analyzed and optimized by an optimizer function that is recursively called over the tree that is representing the expression. The optimizer simplifies expressions which result is known and therefore does not need to be evaluated. For example, it is known that `5 + 3` $\equiv$ `8` and `true && (true || (false && false))` $\equiv$ `true`. When a programmer writes a program, she or he may not want to do all the simple calculations before writing the program in which they appear in, we rely on machines to simplify those processes. Reducing constants be-

fore evalaution may seem unnecessary when writing a small program, but they do take away computation time, and if they appear inside of loops, it is a wise choice to simplify those constant expressions whose result is already known before it is calculated in all the loop iterations. It is also necessary in optimizing programs before compilation. The optimizer, by now, reduces operations between constants and `if` statements whose guard is always true (or false). To achieve minimization to an unreducible form, optimizer calls are repeated until it produces an output equal to its input; this way, we get a tree representing an expression that cannot be optimized again. This process is fairly easy:

```
let rec iterate_optimizer e =
        let oe = optimize e in
        if oe = e then e (* Bottoms
            out *)
        else iterate_optimizer oe
```

5 Types

6 Evaluation

7 Tests

Unit testing is extensively performed using the alcotest testing framework. Code coverage is provided by the `bisect_ppx` library which yields an HTML page containing the coverage percentage when unit tests are run by the dune build system. After each commit is pushed to the remote version control repository on Github, the package is built and tests are run thanks to the Travis Continuos Integration system.

8 Thanks to

- Prof. Gian-Luigi Ferrari for teaching us how to project and develop interpreters in OCaml
- Antonio DeLucreziis for helping me implement lazy evaluation.
- Prof. Alessandro Berarducci for helping me study lambda calculus in deep.