The gobba functional programming language. WORK IN PROGRESS DRAFT

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

January 19, 2020

The gobba logo by Kevin Fontanari

Abstract

gobba is a dynamically typed and purely functional interpreted programming language, heavily inspired from the OCaml, Haskell and Scheme languages. It is based on Professors Gianluigi Ferrari and Francesca Levi's minicaml interpreter example. The goal for gobba is to be a practical language with built in support for scientific computing, solving some of the problems that exist in other dynamically typed interpreted languages like python and Javascript. It features static (lexical scoping), a simple but effective module system, eager and lazy evaluation and an interactive didactical REPL that shows completions and can show insights about each recursive evaluation step.

1 Installing gobba

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

```
opam install gobba
```

Alternatively, **gobba** can be installed from source by downloading the source code repository and building it manually. **gobba** 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/gobba
# cd into the source code directory
cd gobba
# install dependencies
opam install ANSITerminal dune ppx_deriving menhir \\
  cmdliner alcotest bisect_ppx ocamline
# compile
make
# test, execute and install
make test; make run; make install
```

2 Interactive Command Line Interface

If no filename is passed to gobba executable, the interactive shell interface is launched. The REPL shell is built upon a simple library called `ocamline` [3], which itself relies on a library called `ocaml-linenoise` [2] that provides `readline` navigation and completion functionality without additional system dependencies.

3 Syntax and Parser

Lexing is achieved with `ocamllex`, the default tool for generating scanners in OCaml. The parser is realized with the **Menhir** parser generator library [4], and is documented using **Obelisk**, which generates a clean text file containing the language grammar, available in Appendix A.

4 Purity Inference

An important feature of the gobba language is the purity inference algorithm, which is performed statically on expressions before evaluation. It is an interpretation of expressions over the domain of purity, meant to prevent side effects by signal an error if they are contained inside the programs written in the language. Expressions are tagged by the algorithm with the `Pure`, `Impure` and `Numerical` labels. An `Impure` expression is an expression that contains calls to primitives that perform I/O operations, mutable variables and/or imperative style assignments. A `Numerical` expression is an expression where only numerical operations are performed; `Pure` expressions are those which do not fall into the previous two categories.

To achieve the execution of impure side effects, the programmer has two constructs available called **purity blocks**. By default, the evaluator is in an `Uncertain` context, which means that it will not allow side effects to be carried on by evaluation, but will allow evaluating purity blocks that change the currently allowed purity context. The `impure` statement takes an expression (the block) and evaluates it in a context where the allowed purity is `Impure`, so that side effects may be performed. The other construct available, the `pure` statement, takes an expression and enforces a `Pure` context, meaning that side effects and nested impure blocks will not be allowed inside of the expression.

5 AST Optimization

After purity inference is performed, and before evaluation, 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` ≡ `8` and `true && (true || (false && false))` ≡ `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 before evaluation 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
```

Boolean operations are reduced using laws from the propositional calculus, such as DeMorgan's law, complement, absorption and other trivial ones.

6 Types

7 Evaluator

gobba's evaluator is heavily inspired by the Metacircular Evaluator defined in the highly acclaimed textbook *Structure and Interpretation of Computer Programs* [1].

8 Primitives

The language primitives that are implemented in OCaml are organized in modules separated by functionality. Each primitive is a function that accepts a list of evaluated values and returns a single reduced value; therefore they have a type of `evt list -> evt`. OCaml primitives have to perform internal typechecking and unpacking of the arguments they receive from the gobba calls.

From the evaluator's perspective, primitives are organized in a table such that when a symbol gets evaluated, it is looked up in the primitives table, if there is a match then the found primitive's name is wrapped in an `ApplyPrimitive` expression nestedinside of a lazy lambda expression that permits partial application. When the evaluator finally encounters an `ApplyPrimitive` expression, the primitive OCaml function is extracted, applied to the arguments and the resulting value is returned by the current evaluator call. If a primitive is not found when looking up for a symbol, then a symbol lookup is performed in the current environment.

Some primitives, such as catamorphic procedures, are not native OCaml functions but small expressions written directly in gobba; those primitives are kept as lazy expressions into the same table as native OCaml primitives. The key difference between the two resides in the fact that those textual gobba primitives are not transformed into a function which body contains only an `ApplyPrimitive` call, but are instead parsed and analyzed at run time. The resulting additional startup time caused by parsing and analysis is proportional to the number of textual form primitives in the table and therefore quite irrelevant on non-embedded computer systems. The *fold left* and *fold right* catamorphic primitives are written directly in the gobba language and are hereby provided as examples.

Listing 1: The tail recursive left fold procedure

```
fun f z l ->
if typeof l = "list" then
  let aux = fun f z l ->
    if l = [] then z else
    aux f (f z (head l)) (tail l)
  in aux f z l
else if typeof l = "dict" then
  let aux = fun f z kl vl ->
    if kl = [] && vl = [] then z else
    aux f (f z (head vl)) (tail kl) (
       tail vl)
  in aux f z (getkeys l) (getvalues l)
else failwith "value is not iterable"
```

Listing 2: The right fold procedure

```
fun f z l ->
if typeof l = "list" then
   let aux = fun f z l ->
    if l = [] then z else
    f (head l) (aux f z (tail l))
   in aux f z l
else if typeof l = "dict" then
   let aux = fun f z kl vl ->
    if kl = [] && vl = [] then z else
    f (head vl) (aux f z (tail kl) (
       tail vl))
   in aux f z (getkeys l) (getvalues l
      )
else failwith "value is not iterable"
```

9 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.

10 Thanks to

- Prof. Gian-Luigi Ferrari and Francesca Levi for teaching us how to project and develop interpreters in OCaml
- Kevin Fontanari for the pixel art gobba mascotte.
- Antonio DeLucreziis for helping me implement lazy evaluation.
- Prof. Alessandro Berarducci for helping me study lambda calculus in deep.
- Giorgio Mossa for helping me polish the lambda-closure mechanism.

Appendix A Parsing Grammar

```
<optterm_list(separator, X)> ::= [separator]
                               | <optterm_nonempty_list(separator, X)>

<optterm_nonempty_list(separator, X)> ::= X [separator]
                                        | X separator
                                          <optterm_nonempty_list(separator,
                                          X)>

<toplevel> ::= <optterm_nonempty_list(SEMI, <statement>)> EOF

<statement> ::= <ast_expr>
              | <def>
              | <directive>

<assignment> ::= SYMBOL EQUAL <ast_expr>
               | LAZY SYMBOL EQUAL <ast_expr>

<def> ::= LET [<assignment> (AND <assignment>)*]

<directive> ::= DIRECTIVE STRING
              | DIRECTIVE INTEGER
              | DIRECTIVE UNIT

<ast_expr> ::= <ast_app_expr>
             | <ast_expr> CONS <ast_expr>
             | NOT <ast_expr>
             | <ast_expr> BIND <ast_expr>
             | <ast_expr> ATSIGN <ast_expr>
             | <ast_expr> CONCAT <ast_expr>
             | <ast_expr> LAND <ast_expr>
             | <ast_expr> OR <ast_expr>
             | <ast_expr> PLUS <ast_expr>
             | <ast_expr> MINUS <ast_expr>
             | <ast_expr> COMPLEX <ast_expr>
             | <ast_expr> TIMES <ast_expr>
             | <ast_expr> DIV <ast_expr>
             | <ast_expr> EQUAL <ast_expr>
             | <ast_expr> DIFFER <ast_expr>
             | <ast_expr> GREATER <ast_expr>
             | <ast_expr> LESS <ast_expr>
             | <ast_expr> GREATEREQUAL <ast_expr>
             | <ast_expr> LESSEQUAL <ast_expr>
             | IF <ast_expr> THEN <ast_expr> ELSE <ast_expr>
             | <def> IN <ast_expr>
             | LAMBDA SYMBOL+ LARROW <ast_expr>
             | <ast_expr> COMPOSE <ast_expr>
             | <ast_expr> PIPE <ast_expr>

<ast_app_expr> ::= <ast_simple_expr>+

<ast_simple_expr> ::= SYMBOL
                    | UNIT
                    | DOLLAR <ast_expr>
                    | LPAREN <ast_expr> RPAREN
                    | <ast_simple_expr> COLON SYMBOL
                    | PURE <ast_expr>
                    | IMPURE <ast_expr>
                    | LSQUARE <optterm_list(COMMA, <ast_expr>)> RSQUARE
                    | LVECT <optterm_nonempty_list(COMMA, <ast_expr>)> RVECT
                    | LVECT RVECT
```

```
| LBRACKET <optterm_list(COMMA, <assignment>)> RBRACKET
| BOOLEAN
| CHAR
| STRING
| INTEGER
| FLOAT
```

References

[1] H. Abelson, G. J. Sussman, and with Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press/McGraw-Hill, Cambridge, 2nd editon edition, 1996. ISBN 0-262-01153-0.

[2] T. O. community. ocaml-linenoise: Self-contained ocaml bindings to linenoise. URL `https://github.com/ocaml-community/ocaml-linenoise`.

[3] C. J. Nevers. The ocamline library providing a command line interface for user input. URL `https://github.com/chrisnevers/ocamline`.

[4] F. Pottier and Y. Régis-Gianas. Menhir reference manual. *Inria, Aug*, 2019. URL `http://gallium.inria.fr/~fpottier/menhir/manual.html`.