Egel Language Documentation

Release 0.0

M.C.A. (Marco) Devillers

Jan 15, 2019

Contents:

CHAPTER 1

Introduction

Egel is a small toy language based on untyped eager combinator rewriting. Or, equivalently, it is an untyped lambda calculus with constants and strict semantics.

It roughly falls into the same category of combinator languages like Q/Pure and conceptually predates languages like Miranda, ML or Haskell.

The language is homoiconic and supports symbolic rewriting, exceptions, namespaces, and concurrency.

Semantically, the implementation is morally equivalent to an eager term rewriter on a directed acyclic graph with a small twist for performance.

To get a taste of the language an example is shown below.

```
namespace Fibonacci (
    using System

    def fib =
        [ 0 -> 1
        | 1 -> 1
        | N -> fib (N - 2) + fib (N - 1) ]
)

using Fibonacci

def main = fib 5
```

The interpreter is implemented in C++. Sources can be downloaded from Github.

CHAPTER 2

Installation

The Egel interpreter is a small C++ application and is for the moment not distributed in binary form. You will need to download and compile it yourself.

2.1 Building from sources

The interpreter is developed on a Linux system and uses libicu for Unicode support. You need to have GCC/g++, the GNU compiler chain, and the development files for libicu installed. Most Linux package managers will provide that for you.

The sources can be obtained from Github.

To compile the system run the *build.sh* script. That should give you an interpreter named *egel* in the *src* directory and a number of dynamically loadable Egel object files in the *include* directory.

2.2 Installing the interpreter

To install the system run the *install.sh* script as root. On a Fedora system..

If you don't want a system-wide install, please note that you only need the interpreter named *egel* and all files in the *include* directory if you want to do anything useful. You can set the environment variable *EGEL_INCLUDE* to point at the latter path.

2.3 Using the interpreter

A number of example scripts are provided in the examples directory. If you set up your system correctly, you can run any of them with the command *egel example.eg*.

```
user$ egel examples/fib.eg
10946
```

Tip: The interpreter has a REPL, an interactive mode, but doesn't support line editing or completion. I use the console program *rlwrap* for that. It should be installed or be provided by your distribution. To use the interpreter in interactive mode with line editing run the command *rlwrap egel*.

At the prompt of the Egel interpreter you can type small expressions.

```
>> 1 + 2
3
```

However, you'll likely want more functionality. It is recommended you always import the prelude and open the necessary namespaces.

```
>> import "prelude.eg"
>> using System
>> using List
>> foldl (+) 0 {1,2,3}
6
```

CHAPTER 3

The Calculus

Egel is based upon a small combinator calculus with a limited number of constructs. Below, that calculus is introduced with small examples.

3.1 Constants

The smallest Egel expression is a constant.

```
>> 0
0
```

Multiple constants compose.

```
>> 1 2 3
(1 2 3)
```

Egel supports integers, floats, characters and Unicode strings.

```
>> 'a' 3.14 "Hello!"
('a' 3.14 "Hello!")
```

You can define your own constant combinators. Constants are lower-case.

```
>> data one, two, three
>> two
two
```

3.2 The nameless pattern-matching abstraction

The basic work-horse of Egel is the nameless pattern-matching combinator. Roughly similar to an untyped lambda abstraction, where variables are uppercase.

```
>> [ X -> X ] 5
5
```

The nameless pattern-matching combinator may consist of multiple alternatives which pattern match from left to right. You can mix variables and constants in patterns.

```
>> [ 0 -> "zero" | 1 -> "one" | X -> "a lot" ] 1
"one"
```

You can match against multiple values.

```
>> [ X Y -> X - Y ] 4 1
3
```

Caution: Often, you will want to put a space after a - symbol. Can you guess why? It's because constants compose, so *2-1* are the two constants *2* and *-1*. Make sure to insert the space!

You can define combinators as named abstractions for terms.

```
>> def id = [ X -> X ]
>> id "Hi!"
"Hi!"
```

Definitions may mention themselves, then they are recursive.

```
>> def fac = [ 1 -> 1 | N -> N * fac (N - 1) ]
>> fac 3
6
```

Note: If you don't understand the above definition then try replacing *fac* in the term *fac 3*. Like this, *fac 3 = 3 * fac (3 - 1) = 3 * fac 2 = 3 * 2 * fac 1 = 3 * 2 * 1 = 6*. Otherwise, look up 'recursion' on the Internet. Good luck!

Egel refuses to rewrite, or reduce, definitions where none of the patterns matched.

```
>> def z = [ 0 -> 0 ]
>> z 1
(z 1)
```

In the example above, the combinator *z* can only reduce a *0*, when given a *1* as an argument the interpreter refuses to reduce the term.

3.3 Helpful shorthands

With *let/in* you can bind a variable to a value.

```
>> let X = 3 in X + 2
5
```

A condition consists of an *if/then/else* statement.

```
>> if 3 < 5 then "smaller" else "larger"
"smaller"
```

3.4 Exceptions and exception handling

Egel supports exceptions, *throw* any value anywhere.

```
>> 1 + throw "don't go here"
exception("don't go here")
```

You can also catch exceptions in a *try/catch* block. It reduces the try part, any exception thrown in there will handled by the provided catch handler.

```
>> try 1 + throw "don't go here" catch [ E -> "caught:" E ]
("caught:" "don't go here")
```

That's the whole calculus, you can now program in Egel.

CHAPTER 4

Your First Programs

Let's move on and define some programs. An Egel program is a collection of scripts driven by a *main* function.

4.1 Hello World

We'll start with the cannonical example for a new language "hello world". Edit the file *hello.eg* and add the following content.

```
def main = "Hello world!"
```

An Egel script is a text file which may define one *main* function. You don't have to do that, but then it won't run anything.

Run the example with the Egel interpreter.

```
user$ egel hello.eg
Hello world!
```

4.2 More functions

More functions make for more interesting scripts. Let's start of with a function you should know, factorial.

The interpreter starts with a clean slate, you'll need to give a *using System* directive to access the builtin math combinators.

```
using System

def fac = [ 1 -> 1 | N * fac (N - 1) ]

def main = fac 5
```

Save it somewhere, run it, and it should give you a number.

4.3 Adding namespaces

Functions live in namespaces. A namespace starts with a capital letter and names all combinators within the space. Let's move on to the venerable Fibonacci.

```
namespace Fibonacci (

    using System

    def fib =
        [ 0 -> 1
        | 1 -> 1
        | N -> fib (N - 1) + fib (N - 2) ]

)

using Fibonacci

def main = fib 5
```

4.4 Multiple scripts

Of course, you'll want to use and define small data structures. That's easy in Egel, small constants can function as constructors, although when you become more advanced you will often just leave them away.

While we're at it, let's pretend this is serious business and split our new application into two files.

Put the following code in the file *eval.eg*.

```
namespace Eval (

    using System

    data sum, mul

    def eval =
        [ sum X Y -> eval X + eval Y
        | mul X Y -> eval X * eval Y
        | X -> X ]

)
```

And write the following text to *main.eg*.

```
import "eval.eg"

using Eval

def main = eval (sum 3 (mul 2 7))
```

The *import* directive tells the interpreter where to look. Running *egel main.eg* should give *17*.

CHAPTER 5

Lists and Tuples

Right, so you've seen constants, abstractions, and a means to split programs over files. Any pedantic scientist can now tell you that's enough to encode any program. Maybe that's correct, maybe not.

Let's find out.

5.1 Lists

Working with lists follows a convention in Egel. They are constructed with the *nil* and *cons* constants from the *System* namespace. Of course, you are free to choose any convention you like, but for now, we kind-of rely on that programmers will follow that convention.

You can test that the constants are there in interactive mode.

```
>> using System
>> nil
System:nil
```

Creating a list is trivial.

```
>> cons 'a' (cons 1 nil)
(System:cons 'a' (System:cons 1 System:nil))
```

But that's a lot of typing. Egel provides what is called syntactic sugar for lists, a shorthand notation employing curly brackets.

```
>> {'a', 1}
(System:cons 'a' (System:cons 1 System:nil))
```

Let's proceed with defining functions on lists. A length function is the first we'll try.

```
>> def length = [ nil -> 0 | cons X XX -> 1 + length XX ]
>> length {'a', 1}
2
```

Egel is untyped, you might make a typo and apply *length* to something not a list. Can you guess what will happen?

```
>> length 0
(length 0)
```

The patterns are exhausted therefor the term will fail to reduce.

Functional programmers adore lists, there's a lot one can do with them, if not everything. Egel suplies a number of convenience routines in the *List* namespace in the *prelude*.

```
>> import "prelude.eg"
>> using List
```

I'll assume that you know some functional programming. Standardly, we can apply any function *f* to any list with the *map* combinator.

```
>> map [X -> X + 1] {0,1}
(System:cons 1 (System:cons 2 System:nil))
```

This documentation is on the Egel language, it's not an introduction to functional programming. But did you get what happened there? *map* applied *[X->X+1]* to both elements of the list *{0,1}* resulting in the list *{1,2}*.

And the important *foldl* is defined too. It's a useful operator but don't go overboard with it!

```
>> foldl (+) 0 {1,2,3}
6
```

foldl will fold a function and a constant over a list, *foldl (+) 0 {1,2,3} = 1 + (2 + (3 + 0))*. It's a summation.

5.2 Tuples

Tuples in languages are used to group things. It's a useful feature which you don't always need in Egel since constants compose. Let's find out how they work.

Like lists, tuples are syntactic sugar for applying the *tuple* constant out of the *System* namespace to a number of arguments.

```
>> (1,"hi")
(System:tuple 1 "hi")
```

Again, it's all untyped so we can try to match against a tuple to find out how many fields it has.

```
>> def c = [ (X,Y) -> 2 | (X,Y,Z) -> 3 ]
>> c ("what", "a", "night")
3
```

That's all for that subject. If you start programming Egel you'll find many more useful constructs.

Note: Egel has a concise syntax, so you might easily get confused between alternatives.

The folowing reduces two arguments. Two patterns, each one variable.

```
>> [X Y -> X] 0 1
0
```

And this rewrites two composed constants. One pattern of two variables.

```
>> [(X Y) -> X] (0 1)
0
```

And finally, this rewrites a tuple. One pattern using sugar for a tuple.

```
>> [(X, Y) -> X] (0, 1)
0
```

CHAPTER 6

Conway's Game of Life

To showcase that Egel can be used to write real programs I'll walk you through an example of a small Conway's life application. I'll assume you know some functional programming, some territory not covered yet comes along too.

Conway's game of life plays on a grid. At any point a cell on the grid may be dead or alive. Any life cell with fewer than two, or more than three, neighbours dies. Any dead cell with exactly three neighbours comes alive.

6.1 Preamble

It's good practice to start every file with some comment on what it implements.

```
####
# Conway's Game of Life.
#
```

Like a lot of languages, single-line comments start with *#*.

We'll rely on combinators defined in *prelude.eg* and *io.ego*. A *.ego* file is an object file, a binary on your system.

```
import "prelude.eg"
import "io.ego"
```

We'll open up the different namespaces we need from those files.

```
using System
using List
using IO
```

6.2 The board

The board size of the two-dimensional grid is defined with a constant.

```
def boardsize = 5
```

So, now the real programming starts. The grid is implemented as a stencil. A stencil is a function mapping coordinates to cells. A cell *0* is dead, any other value means it's alive.

The empty grid maps all coordinates to dead cells.

```
def empty = [ X Y -> 0 ]
```

To insert an alive cell, we update the stencil with a clause mapping two matching coordinates to an alive cell.

```
def insert =
    [ X Y BOARD ->
        [ X0 Y0 -> if and (X0 == X) (Y0 == Y) then 1
                    else BOARD X0 Y0 ] ]
```

To get to all coordinates we map multiple times on the list *{0,..,boardsize-1}* to retrieve the pairs *{{0 0, 0 1, ..},..}*. Note that we don't need to tuple explicitly.

```
def coords =
    let R = fromto 0 (boardsize - 1) in
        [ XX YY -> map (\X -> map (\Y -> X Y) YY) XX ] R R
```

6.3 Printing

To print, we just apply *IO.print* for a dead or alive cell. Though Egel is a mostly pure term rewrite system, combinators loaded may have side effects.

```
def printcell =
    [ 0 -> print ". "
    | _ -> print "* " ]
```

A wildcard pattern _ is used to match against any value.

Printing a board is done by going over all coordinates and printing the cell for that coordinate.

```
def printboard =
    [ BOARD ->
        foldl [_ XX -> map [(X Y) -> printcell (BOARD X Y)] XX; print "\n" ] nop␣
↪coords ]
```

Note: Though Egel combinators may be side-effecting, they must reduce to a value. *IO:print* will print all its arguments but will reduce to the uninformative value *System:nop*. Often, with side-effecting calculations these values are simply discarded. The semicolon separates such statements.

6.4 Generations

The neighbour count of a coordinate on a board can be calculated by just looking around.

```
def count =
    [ BOARD X Y ->
        (BOARD (X - 1) (Y - 1)) + (BOARD (X) (Y - 1)) + (BOARD (X+1) (Y - 1)) +
        (BOARD (X - 1) Y) + (BOARD (X+1) Y) +
        (BOARD (X - 1) (Y+1)) + (BOARD (X) (Y+1)) + (BOARD (X+1) (Y+1)) ]
```

The status of the next cell is calculated from whether the current cell is alive or dead and the number of neighbours.

```
def next =
    [ 0 N -> if N == 3 then 1 else 0
    | _ N -> if or (N == 2) (N == 3) then 1 else 0 ]
```

A board is updated by applying the above function *next* to every coordinate on the board.

```
def updateboard =
    [ BOARD ->
        let XX = map (\(X Y) -> X Y (BOARD X Y) (count BOARD X Y)) (flatten coords) in
        let YY = map (\(X Y C N) -> X Y (next C N)) XX in
            foldr [(X Y 0) BOARD -> BOARD | (X Y _) BOARD -> insert X Y BOARD ] empty␣
↪YY ]
```

6.5 A blinker

A blinker consists of three alive cells next to each other.

```
def blinker =
    (insert 1 2) . (insert 2 2) . (insert 3 2)
```

We print three generations of a board with a blinker.

```
def main =
    let GEN0 = blinker empty in
    let GEN1 = updateboard GEN0 in
    let GEN2 = updateboard GEN1 in
        foldl [_ G -> print "generation:\n"; printboard G ] nop {GEN0, GEN1, GEN2}
```

And that wraps it up. A real Egel application.

CHAPTER 7

Concurrency

Because Egel is a term rewrite language, it is trivial to rewrite terms in parallel. Concurrency is provided through a combinator.

7.1 Parallel rewriting

If you want to have two computations run in parallel use the *par* combinator. It takes two abstractions to be reduced, applies both of them to a dummy argument, and returns a tuple containing both results.

```
>> using System
>> par [ _ -> 1 + 2 ] [ _ -> 3 + 4 ]
(System:tuple 3 7)
```

Note: The *par* combinator takes two abstractions because Egel has strict semantics. If it would have been just *par (1+2) (3+4)* the interpreter would have reduced the arguments to *par* first, resulting in the parallel reduction of *par 3 7*. By wrapping the computations their evalution is deferred.

We can inspect what arguments are given to the abstractions of *par*.

```
>> using System
>> par [ X -> X ] [ X -> X ]
(System:tuple System:nop System:nop)
```

Hardly interesting.

Of course, you might want to supply arguments to both terms to be reduced. Then simply wrap them in an abstraction again.

```
>> using System
>> [ X -> par [ _ -> X * 3 ] [ _ -> X + 5 ] ] 4
(System:tuple 12 9)
```

7.2 Parallel Fibonacci

With all what we know now, we can implement parallel Fibonacci.

```
import "prelude.eg"

namespace Fibonnaci (
  using System

  def pfib =
    [ 0 -> 0
    | 1 -> 1
    | X -> [ (F0, F1) -> F0 + F1 ] (par [_ -> pfib (X - 1) ] [_-> pfib (X - 2)]) ]

)

using Fibonnaci
using System

def main = pfib 10
```

In the recursive alternative of *pfib* it will start up two parallel computations, reduce those, after which it will deconstruct the pair returned and add both components.

Nifty, huh?

Caution: Though morally Egel could support cheap concurrency, the *par* combinator is implemented with the C++ thread library, thus with system threads.

System threads are a bit heavyweight and easy to run out of. On my machine, I can start upto roughly 20,000 threads. Go easy on *pfib*!

CHAPTER 8

Discussion

Egel is a toy language implementing an eager untyped combinator calculus as a term rewriting system on a directed acyclic graph (DAG) through lifting C++.

What does that mean? You can easily conclude a number of things from that although you can discuss those conclusions endlessly.

1. Egel is untyped. That means a lot but among others that it doesn't scale very well, though Lisp, Javascript, and Python practitioners might disagree with that. Types are great, however, for short programs they don't matter much, with types you just pay a little for static guarantees.
2. Terms are rewritten. Well, that's likely not a fast language and although the interpreter gives reasonable performance it indeed isn't fast. Though, through Herculean effort, term rewrite systems can be made performant, I don't have that much time. However, the interpreter gives you a pretty robust system and that's worth something too.
3. It rewrites a DAG. Right, no mutation since that would mean you could introduce cycles. The interpreter implements some unsafe extensions which ameliorate that a bit but in principle you don't have access to that. The language is Turing complete, however, you'll need to be an avid functional programmer. I hope that going with a DAG will pay off in the distant future since it trades off global analysis for local analysis during garbage collection.
4. C++. That's another tradeoff. C++ objects are heavyweight so you pay again in performance but you get a bit more reliable system back. The good part is that it is relatively easy to safely drop C++ functionality into combinators.

In conclusion, Egel is a solution for people who need a small declarative easily extendable language which effortlessly binds to C/C++ and who don't expect to write very large or imperative programs. It tries to support a niche market.

Apart from that, you can have great fun writing Egel programs so don't let any of the above stop you!

CHAPTER 9

Thanks for helping out!

I would like to thank the following people.

Athas@freenode.net for running *#proglangdesign* and trying to make the interpreter run on FreeBSD with clang. No luck so far, but that might change!

mahmudov@freenode.net who built the interpreter for the first time on another machine and added it to the Milis Linux distribution.

lijero@freenode.net for wise comments.

Come visit on channel *#egel* on *irc.freenode.net*.

CHAPTER 10

Indices and tables

- genindex
- search