Egel — Graph Rewriting with a Twist

M.C.A. (Marco) Devillers
marco.devillers@gmail.com

ABSTRACT

Egel is an untyped eager combinator toy language. Its primary purpose is to showcase an abstract graph-rewriting semantics allowing a robust memory-safe construction in C++. Though graph rewriters are normally implemented by elaborate machines, this can mostly be avoided by changing the representation of a term graph. With an informal inductive argument the resulting representation is shown to always form a directed acyclic graph. Moreover, this graph semantics can trivially be extended to allow exception handling and cheap concurrency. Egel, the interpreter, exploits this semantics with a straight-forward mapping from combinators to reference counted C++ objects.

1. INTRODUCTION

It all started with Lisp. Except that it didn't. Throughout history, people have been interested in mechanizing math and, more recently, mathematical approaches to programming. Countless researchers have contributed to this ideal, most are forgotten, but certain influential milestones can be identified which tell a story from symbolic evaluation to graph-driven combinatorial rewriting.

Lisp[3] put the representation and symbolic evaluation of expressions first and coupled that with a versatile operational semantics; the language and ideas behind it remain influential to this day. The first work on the mechanical evaluation of non-strict languages was laid down by Landin[2] resulting research which put closures first. Turner's work on SASL[5] diverged from that and concentrated on SK-combinator-driven evaluation culminating in the typed and lazily reduced Miranda[6]. Combinator-driven lazy graph rewriting spurred a number of abstract machines such as the Spineless Tagless Graph Machine[4] behind Haskell and the Parallel ABC Machine[1] for CLean.

But while graph rewriting is a pleasingly clean means to give an operational semantics to a term-rewriting language, ultimately it was deemed to slow and compilers for functional languages now usually invest a great deal into compiling to more traditional schemes. However, because graph-rewriting is such a simple model with some exceptional properties, it allows for trivialized implementations of term-rewriting languages.

Egel exploits a novel view on eager graph rewriting to implement a term-rewriting language in a robust and memory-safe manner in C++, at the cost of performance.

2. GRAPH REWRITING

The notion of graph rewriting starts with the observation that usually a term of a language can be given a pictorial representation. In figure 1, the traditional tree representation of the term `mul (1 + 2) (inc 1)`, a running example, is given. The `@` node depicts application.

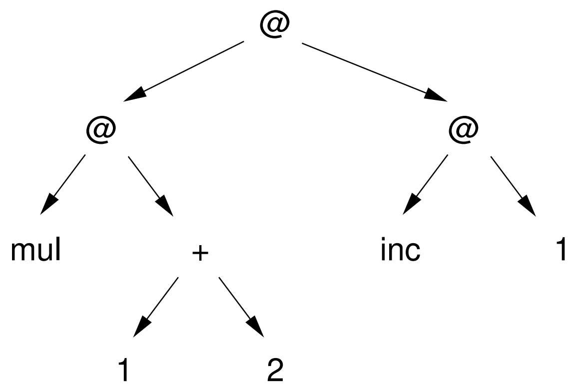

Figure 1: standard graphical representation

However, the representation of a term graph in computer memory is often slightly different. In figure 2, a standard 'thunked' representation of `mul (1 + 2) (inc 1)`, a thunk is an array of pointers to constants and combinators. Note that the `@` application node is gone conforming to that storing unnecessary application nodes would be too costly regarding both storage and performance.

Given the thunked representation of term graphs a straight-forward approach towards an evaluator would be to introduce primitives and graph-manipulation code for combinators combined with a stack machine which holds redexes to rewrite, traces of that can be found in both the G-Machine and the PABC machine.

Instead of that, Egel terms are compiled to a twisted representation, as shown in figure 3, bypassing the need for a stack. Thunks are extended at the front with two pointers, one pointer points to what to do –rewrite– next and another pointer where to store the result. The `*` root node points to what will be rewritten first.

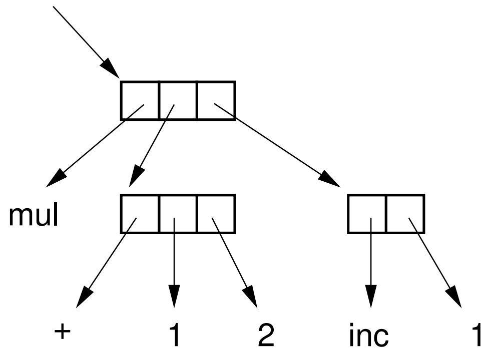

Figure 2: thunked representation

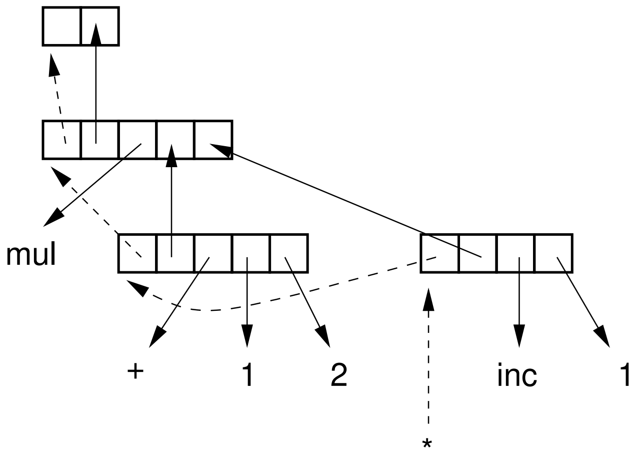

Figure 3: twisted representation

The cost of allocating extra pointers in a thunk may seem wasteful but shouldn't be more burdensome than allocating a thunk and some stack space. Though the benefits of this approach will be completely undone by Egel's idiomatic C++ implementation.

The chain of redexes to rewrite makes explicit that the reduction order is strict or eager. Arguments to functions are rewritten first, in right-to-left order, after which the function is applied. Reduction is performed by repeatedly rewriting the top root pointer.

Figure 4 shows the term after the first two arguments of `mul` are rewritten. Note that reduced arguments are always pointed towards; i.e., should form a tree.

The fully reduced term is shown in figure 5. At this point the runtime can be called by the root rewriting pointer and might, for instance, print the result.

The clue of this paper: Where we originally started of with a tree, or with sharing a directed acyclic tree (DAG), each other figure still is a tree, or DAG. Changing representation or rewriting kept this invariant, which is a feature exploited to implement this scheme on top of native C++ reference counted objects.

3. THE EGEL LANGUAGE

The Egel language is an experimental front-end with the

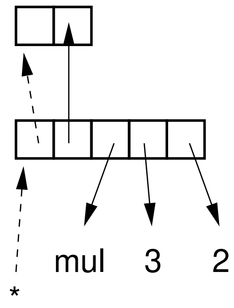

Figure 4: after evaluation of two arguments

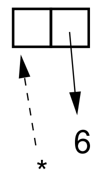

Figure 5: final term

previously described graph semantics. It's not thoroughly discussed here, with two examples just enough of a taste of the language is given to understand a follow-up informal argument. Below, an example Egel script implementing the Fibonacci function.

```
import "prelude.eg"

namespace Fibonacci (
    using System

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

using Fibonacci

def main = fib 5
```

Superficially, Egel isn't much different from other functional programming languages. As shown, scripts can include other scripts, it has namespaces, and definitions of functions may be recursive.

What is slightly different is that functions are defined with guarded lambda abstractions which are directly mapped to (unnamed) combinators by the interpreter.

The example script below shows how lists are defined and used.

```
namespace List (
    data nil, cons

    def ++ =
        [ nil YY -> YY
        | (cons X XX) YY -> cons X (XX ++ YY) ]
)
```

Noteworthy is that a good approximation of Egel is to think of it as a lambda calculus with constants where constants may compose; e.g., `(1 2)` is a legal term in Egel.

That feature is exploited to introduce the notion of lists; in the example script two constants **`nil`** and **`cons`** are defined which, as any other constant, may be applied to any number of arguments. Guarded abstractions are then used to define (recursive) functions which may decompose their arguments.

4. DIRECTED ACYCLIC GRAPH PROPERTY

This section discusses the heart of this paper, an informal argument.

THEOREM 4.1. *Egel terms in the runtime always form a tree, or directed acyclic graph.*

This is at heart an inductive argument which relies on a property of the front-end Egel language. From now on, tree is written where we also mean directed acyclic graph.

LEMMA 4.2. *Fully reduced expressions always form a tree.*

This is fundamentally a property of the front-end language since combinators could, in principle, rewrite terms in the runtime to anything. However, it is assumed that combinators are the result of the translation of code in the front end, i.e., complex expressions of guarded anonymous abstractions. Since abstractions can only take apart and reassemble complex trees, with some confidence this property holds.

LEMMA 4.3. *The chain of redexes always forms a tree, even during rewriting.*

This is an inductive argument. First, the base step, the intial term **`main`** forms a tree, which is trivially true by inspection.

Then, the inductive step, if the chain of redexes forms a tree, then rewriting won't change that. This holds because a rewrite can result in either of two things: Either the fully reduced result (a tree) is placed in a receiving thunk and rewriting proceeds with the next redex, and that is trivially again a tree. Or, the chain of redexes is expanded with a new number of redexes conforming to the right-hand-side of a guarded abstraction, and that must form a tree.

5. C++

6. CONCLUSIONS

7. REFERENCES

[1] T. H. Brus, M. C. J. D. van Eekelen, M. O. van Leer, and M. J. Plasmeijer. Clean – a language for functional graph rewriting. In *Functional Programming Languages and Computer Architecture*, pages 364–384, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg.

[2] P. Landin. The mechanical evaluation of expressions. In *The Computer Journal*, volume 6, pages 308–320, January 1964.

[3] J. McCarthy, R. Brayton, D. Edwards, P. Fox, L. Hodes, D. Luckham, K. Maling, D. Park, and S. Russell. *Lisp I Programmer's Manual.* Cambridge, Massachusetts, March 1960.

[4] S. Peyton Jones. Implementing lazy functional languages on stock hardware: The spineless tagless g-machine. *Journal of Functional Programming*, 2:127–202, 1992.

[5] D. Turner. An implementation of sasl. Technical Report TR/75/4, Department of Computer Science, Colorado State University, 1975.

[6] D. Turner. An overview of miranda. *SIGPLAN Notices*, 21(12):158–166, December 1986.