Conq: A Perl to C compiler

Hugo Kapp

November 18, 2018

Contents

Introduction

Perl is a famous scripting language, mostly used for text handling. It provides regular expressions as part of its core elements, and permits easy rewriting of text files, loading of config files, csv building / extraction ...

Perl is an interpreted language. The main reason for this is the fact that it is a scripting language in the pure tradition of sh or python. This means that it is dynamically typed, and runtime checks must be performed to figure out the semantics of every basic operator. This is easier done in an interpreter than in machine code.

Having an interpreter has the disadvantage of being slow when executing regular expression matching. This gives poor performance for regular expression heavy programs, which is supposed to be the core of Perl.

We describe an alternative, called Conq. Conq is a Perl to C compiler. This means we can compile Perl programs down to machine code, improving the performance of regular

expression matching. Later, we may also investigate runtime specialization to address the performance issues related to runtime type dispatch.

1 Goals

1.1 Efficient regular expression matching

The main point we want to address is the translation from regular expression to efficient machine code. We value this aspect for the following reasons :

- This is the part where we believe we can achieve the biggest improvement over the current Perl interpreter
- This is the most interesting part to work on

1.2 Runtime specialization

An orthogonal point we could investigate is linked to runtime specialization. There are indeed a lot of room for optimization with runtime specialization in Perl scripts.

One case of such optimizations opportunities is regarding variables and operator typing. Due to the dynamically-typed nature of Perl, each operator must, at runtime, check which types its operand are, and apply the correct semantics with regard to this type. This means that, if $a = a+b$ must be executed, then in the case where a and b are both integers the program would perform addition, but if a and b are strings, then the operation is a concatenation. This typing information is only available at runtime.

Lots of things can be done in the case described above. First, we can profile the input types to this + operator. If they are always the same, we can replace the operator runtime checks by the actual code for the correct semantics.

We can also profile the types of the variables a and b. If they never change, we can actually replace the code of every operator using a and b.

Another place for runtime specialization in Perl scripts comes from dynamic regular expressions. Consider the following program : $/aab\$d/$. Different scenarios may arise from the previous example :

1. d is a program constant
   The regular expression can be compiled down to machine code
   This can be done with static analysis
2. d is a per-execution constant
   The regular expression can be compiled down to machine code for each execution of the program (only useful if executed more than once)
   This can be found out statically, but the code must be generated at runtime
3. d is not constant, but always evaluates to the same value at runtime
   This can only be found out at runtime using profiling
   Machine code (with guards) can be generated at runtime
4. d may have multiple values at runtime
   Nothing can be done. A generic and inefficient version must be generated ahead of time

Case 1 can be done completely ahead of time. Case 2 must generate the code at runtime, but the information can already be given ahead of time. Case 3 must be done at runtime (not guaranteed to yield results though).

The lifespan of these specializations is not determined yet. It may either be :

- Per execution
  This has the advantage of reflecting the types of the currently executing program, but must be reexecuted everytime and makes the specialization process hard to write (code rewriting during execution)

- Per script
  In this variant, runtime information is gathered during execution, and specialized code generated at the end of the execution, then recompiled
  This makes the specialization process easier to write, and allows subsequent executions of the same scripts to be faster right away.

Because it is simpler to write and makes a lot of sense (we believe variables used by the programmer always have the same meaning between executions, i.e. always the same type), we might rather go for the second option.

Generally, we consider runtime specialization to be a late optimization in the development process.

1.3 Provide a viable replacement for Perl

Eventually, this means providing all the options Perl does. The most important here might be the in-line script definition, i.e. `perl -e 's/aa/a/g'`.

One other important aspect of Perl we must provide is the reading from files (also from stdin).

1.4 Roadmap for language support

The roadmap for language support should be the following :

1. Regular expressions
   (a) Simple regular expression matching
   (b) Static replacements
   (c) Complete regular expression matching
   (d) Dynamic replacements
2. Scripting
   (a) Global variables
   (b) Variables assigned regular expression results
   (c) Printing
   (d) Basic operators
   (e) Control flow
   (f) stdin handling
   (g) File handling

2 Regular expressions to DFA

There is already theory about how to represent a regular expression as a DFA. It is easier to turn the regexp into a NFA, and then use basic rules to transform the latter into a DFA.

2.1 Regular expressions to NFA

We provide here a recap of how the various elements of a regular expression can be transformed into an NFA.

2.2 From NFA to DFA

We provide here the simple rules that can be used to turn an NFA into a DFA.

2.3 From regular expression directly to DFA

Using the rules defined in the sections above, we give here a direct translation from the basic elements of regular expressions into a DFA. This is what is used in our system.

3 Regular expression compilation

We lay out in this section the design for the compilation of regular expressions.

3.1 Parsing

3.2 Intermediate Representation

We detail here the representation we use for DFAs inside the system, i.e. our IR.

3.3 DFA to C

3.4 Backtracking

Backtracking must be used when a regular expression specifies an alternative. The first alternative is tested for match, and if it does not, then backtracking must be performed on the stream to test the second alternative.

3.5 Filling regexp variables

Regular expressions define variables that represent a subset of the matched string. We describe here how these are managed in Conq.

4 Scripts compilation

5 Discussion points

5.1 Source and destination language

5.1.1 Source language

We have two options here :

1. Start from source Perl scripts
2. Compile the already anayzed Perl bytecode

	Perl scripts	Perl bytecode
Advantages	All information about the program	Already compiled and analyzed source code Can be integrated in the current Perl interprete
Disadvantages	Complex parser	No need to reimplement the parser

5.1.2 Destination language

The destination language will mostly depend on the goals pursued. It can be either :

- A low-level language easily compiled down to machine code (most likely C, could be LLVM)
- Truffle

If we want to support a lot of runtime specializations, then it may be easier to go for Truffle (though we might loose on the regular expression part).

On the other hand, writing the runtime specializations by hand is a great learning exercise.

5.2 Simple runtime specialization

Perl:

```
a = a + b;
```

C:

```
lasttypea;
lasttypeb;
if (a.type == lasttypea \&\& b.type == lasttypeb)
  goto lastjump
else if (a.type == int \&\& b.type == int) {
:plusint
  a = a + b;
  lasttypea=int;
  lasttypeb=int;
  lastjump = plusint;
}
else if (...) {
...
```

5.3 Typed operation dispatch

Perl:

```
a = a + b;
```

C:

```
pluslabels gotolabels[][] = ...;
goto pluslabels[a.typecode][b.typecode];
```