A Play on Regular Expressions

Functional Pearl

Sebastian Fischer Frank Huch Thomas Wilke

Christian-Albrechts University of Kiel, Germany
{sebf@,fhu@,wilke@ti.}informatik.uni-kiel.de

Abstract

Cody, Hazel, and Theo, two experienced Haskell programmers and an expert in automata theory, develop an elegant Haskell program for matching regular expressions: (i) the program is purely functional; (ii) it is overloaded over arbitrary semirings, which not only allows to solve the ordinary matching problem but also supports other applications like computing leftmost longest matchings or the number of matchings, all with a single algorithm; (iii) it is more powerful than other matchers, as it can be used for parsing every context-free language by taking advantage of laziness.

The developed program is based on an old technique to turn regular expressions into finite automata which makes it efficient both in terms of worst-case time and space bounds and actual performance: despite its simplicity, the Haskell implementation can compete with a recently published professional C++ program for the same problem.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; F.1.1 [*Computation by Abstract Devices*]: Models of Computation (Automata)

General Terms Algorithms, Design

Keywords regular expressions, finite automata, Glushkov construction, purely functional programming

Cast

Cody – proficient Haskell hacker

Hazel – expert for programming abstractions

Theo – automata theory guru

Act I

Scene I. Specification

To the right: a coffee machine and a whiteboard next to it.

ICFP'10, September 27–29, 2010, Baltimore, Maryland, USA.

To the left: Hazel *sitting at her desk, two office chairs nearby, a whiteboard next to the desk, a laptop, keyboard, and mouse on the desk.* Hazel *is looking at the screen, a browser window shows the project page of a new regular expression library by Google.*

Cody *enters the scene.*

Cody What are you reading?

Hazel Google just announced a new library for regular expression matching which is—in the worst case—faster and uses less memory than commonly used libraries.

Cody How would we go about programming regular expression matching in Haskell?

Hazel Well, let's see. We'd probably start with the data type. (*Opens a new Haskell file in her text editor and enters the following definition.*)

```
data Reg = Eps              -- ε
         | Sym Char         -- a
         | Alt  Reg Reg     -- α|β
         | Seq  Reg Reg     -- αβ
         | Rep  Reg         -- α*
```

Theo (*a computer scientist, living and working three floors up, strolls along the corridor, carrying his coffee mug, thinking about a difficult proof, and searching for distraction.*) What are you doing, folks?

Hazel We just started to implement regular expressions in Haskell. Here is the first definition.

Theo (*picks up a pen and goes to the whiteboard.*) So how would you write

$$((a|b)^*c(a|b)^*c)^*(a|b)^* \ ,$$

which specifies that a string contains an even number of c's?

Cody That's easy. (*Types on the keyboard.*)

```
ghci> let nocs = Rep (Alt (Sym 'a') (Sym 'b'))
ghci> let onec = Seq nocs (Sym 'c')
ghci> let evencs = Seq (Rep (Seq onec onec)) nocs
```

Theo Ah. You can use abbreviations, that's convenient. But why do you have Sym in front of every Char?— That looks redundant to me.

Hazel Haskell is strongly typed, which means every value has exactly one type! The arguments of the Alt constructor must be of type Reg, not Char, so we need to wrap characters in the Sym constructor.— But when I draw a regular expression, I leave out Sym, just for simplicity. For instance, here is how I would draw your expression. (*Joins* Theo *at the whiteboard and draws Figure 1.*)

Cody How can we define the language accepted by an arbitrary regular expression?