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.

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?

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

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

Copyright © 2010 ACM 978-1-60558-794-3/10/09...$10.00

Seq
Rep Rep
Seq Alt
Seq Seq
Rep Rep
Alt Alt
'a' 'b' 'c' 'a' 'b' 'c' 'a' 'b'

Figure 1. The tree representation of the regular expression $((a|b)^*c(a|b)^*c)^*(a|b)^*$ which matches all words in $\{a,b,c\}^*$ with an even number of occurrences of c

THEO As a predicate, inductively on the structure of your data type. (*Writes some formal definitions to the whiteboard: semantic brackets, Greek letters, languages as sets, etc.*)

HAZEL (*goes to the keyboard, sits down next to* CODY.) Ok, this can be easily coded in Haskell, as a characteristic function. List comprehensions are fairly useful, as well.
(*Writes the following definition in her text editor.*)

```
accept :: Reg → String → Bool
accept Eps        u = null u
accept (Sym c)    u = u == [c]
accept (Alt p q)  u = accept p u ∨ accept q u
accept (Seq p q)  u =
  or [accept p u₁ ∧ accept q u₂ | (u₁, u₂) ← split u]
accept (Rep r)    u =
  or [and [accept r uᵢ | uᵢ ← ps] | ps ← parts u]
```

THEO Let me see. split produces all decompositions of a string into two factors, and parts stands for "partitions" and produces all decompositions of a string into an arbitrary number of factors.

CODY Wait! We need to be careful to avoid empty factors when defining parts. Otherwise there is an infinite number of possible decompositions.

HAZEL Right. But split must also produce empty parts and can be defined as follows. (*Continues writing the Haskell program.*)

```
split :: [a] → [([a], [a])]
split []       = [([], [])]
split (c : cs) = ([], c : cs) : [(c : s₁, s₂) | (s₁, s₂) ← split cs]
```

The function parts is a generalization of split to split words into any number of factors (not just two) except for empty ones.

CODY That's tricky. Let's use list comprehensions again. (*Sits down on one of the empty chairs, grabs the keyboard and extends the program as follows:*)

```
parts :: [a] → [[[a]]]
parts []       = [[]]
parts [c]      = [[[c]]]
parts (c : cs) =
  concat [[(c : p) : ps, [c] : p : ps] | p : ps ← parts cs]
```

We split a word with at least two characters recursively and either add the first character to the first factor or add it as a new factor.

THEO Why do you write [a] and not String.

HAZEL That's because we want to be more general. We can now work with arbitrary list types instead of strings only.

THEO That makes sense to me.

CODY Maybe, it's good to have a separate name for these lists. I think Hazel used the term words—that's a good term. Let's stick to it.

THEO I want to check out your code. (*Sits down as well. Now all three build a small crowd in front of the monitor.*)

```
ghci> parts "acc"
[["acc"],["a","cc"],["ac","c"],["a","c","c"]]
ghci> accept evencs "acc"
True
```

THEO Aha. (*Pauses to think for a moment.*) Wait a second! The number of decompositions of a string of length $n+1$ is 2^n. Blindly checking all of them is not efficient. When you convert a regular expression into an equivalent finite-state automaton and use this automaton for matching, then, for a fixed regular expression, the run time of the matching algorithm is linear in the length of the string.

HAZEL Well, the program is not meant to be efficient. It's only a specification, albeit executable. We can write an efficient program later. What I am more interested in is whether we can make the specification a bit more interesting first. Can it be generalized, for instance?

THEO (*staring out of the window.*) We can add weights.

HAZEL Weights?

SCENE II. WEIGHTS

HAZEL, CODY, *and* THEO *are still sitting around the laptop.*

HAZEL What do you mean by weights?

THEO Remember what we did above? Given a regular expression, we assigned to a word a boolean value reflecting whether the word matches the given expression or not. Now, we produce more complex values—semiring elements.

HAZEL What's an example? Is this useful at all?

THEO A very simple example is to determine the length of a word or the number of occurrences of a given symbol in a word. A more complicated example would be to count the number of matchings of a word against a regular expression, or to determine a leftmost longest matching subword.

CODY That sounds interesting, but what was a semiring, again?

HAZEL If I remember correctly from my algebra course a semiring is an algebraic structure with zero, one, addition, and multiplication that satisfies certain laws. (*Adds a Haskell type class for semirings to the Haskell file.*)

```
class Semiring s where
  zero, one :: s
  (⊕), (⊗) :: s → s → s
```

Here, zero is an identity for ⊕, one for ⊗, both composition operators are associative, and ⊕ is commutative, in addition.

THEO That's true, but, moreover, the usual distributivity laws hold and zero annihilates a semiring with respect to multiplications, which means that both zero ⊗ s and s ⊗ zero are zero for all s.

HAZEL These laws are not enforced by Haskell, so programmers need to ensure that they hold when defining instances of Semiring.

CODY Ok, fine. I guess what we need to do is to add weights to the symbols in our regular expressions.

THEO (*sipping coffee*) Right.

CODY So let's make a data type for weighted regular expressions.

THEO (*interjects.*) Cool, that's exactly the terminology we use in formal language theory.

CODY The only change to what we had before is in the symbol case; we add the weights. We can also generalize from characters to arbitrary symbol types. (*Writes the following code.*)

```
data Reg_w c s = Eps_w
               | Sym_w (c → s)
               | Alt_w (Reg_w c s) (Reg_w c s)
               | Seq_w (Reg_w c s) (Reg_w c s)
               | Rep_w (Reg_w c s)
```

HAZEL Aha! A standard implementation for the function attached to some character would compare the character with a given character and yield either zero or one:

```
sym :: Semiring s ⇒ Char → Reg_w Char s
sym c = Sym_w (λx → if x == c then one else zero)
```

Using sym, we can translate every regular expression into a weighted regular expression in a canonical fashion:

```
weighted :: Semiring s ⇒ Reg → Reg_w Char s
weighted Eps       = Eps_w
weighted (Sym c)   = sym c
weighted (Alt p q) = Alt_w (weighted p) (weighted q)
weighted (Seq p q) = Seq_w (weighted p) (weighted q)
weighted (Rep p)   = Rep_w (weighted p)
```

THEO How would you adjust accept to the weighted setting?

HAZEL I replace the Boolean operations with semiring operations. (*Goes on with entering code.*)

```
accept_w :: Semiring s ⇒ Reg_w c s → [c] → s
accept_w Eps_w       u = if null u then one else zero
accept_w (Sym_w f)   u = case u of [c] → f c; _ → zero
accept_w (Alt_w p q) u = accept_w p u ⊕ accept_w q u
accept_w (Seq_w p q) u =
  sum [accept_w p u_1 ⊗ accept_w q u_2 | (u_1, u_2) ← split u]
accept_w (Rep_w r)   u =
  sum [prod [accept_w r u_i | u_i ← ps] | ps ← parts u]
```

THEO How do you define the functions sum and prod?

HAZEL They are generalizations of or and and, respectively:

```
sum, prod :: Semiring s ⇒ [s] → s
sum  = foldr (⊕) zero
prod = foldr (⊗) one
```

And we can easily define a Semiring instance for Bool:

```
instance Semiring Bool where
  zero = False
  one  = True
  (⊕)  = (∨)
  (⊗)  = (∧)
```

THEO I see. We can now claim for all regular expressions r and words u the equation accept r u == accept_w (weighted r) u.

CODY Ok, but we have seen matching before. Theo, can I see the details for the examples you mentioned earlier?

THEO Let me check on your algebra. Do you know any semiring other than the booleans?

CODY Well, I guess the integers form a semiring. (*Adds a corresponding instance to the file.*)

```
instance Semiring Int where
  zero = 0
  one  = 1
  (⊕)  = (+)
  (⊗)  = (∗)
```

THEO Right, but you could also restrict yourself to the non-negative integers. They also form a semiring.

HAZEL Let's try it out.

```
ghci> let as = Alt (Sym 'a') (Rep (Sym 'a'))
ghci> accept_w (weighted as) "a" :: Int
2
ghci> let bs = Alt (Sym 'b') (Rep (Sym 'b'))
ghci> accept_w (weighted (Seq as bs)) "ab" :: Int
4
```

It seems we can compute the number of different ways to match a word against a regular expression. Cool! I wonder what else we can compute by using tricky Semiring instances.

THEO I told you what you can do: count occurrences of symbols, determine leftmost matchings, and so on. But let's talk about this in more detail later. There is one thing I should mention now. You are not right when you say that with the above method one can determine the number of different ways a word matches a regular expression. Here is an example. (*Uses again the interactive Haskell environment.*)

```
ghci> accept_w (weighted (Rep Eps)) "" :: Int
1
```

CODY The number of matchings is infinite, but the program gives us only one. Can't we fix that?

THEO Sure, we can, but we would have to talk about closed semirings. Let's work with the simple solution, because working with closed semirings is a bit more complicated, but doesn't buy us much.

HAZEL (*smiling*) The result may not reflect our intuition, but, due to the way in which we defined parts, our specification does not count empty matchings inside a repetition. It only counts one empty matching for repeating the subexpression zero times.

ACT II

Same arrangement as before. The regular expression tree previously drawn by CODY, *see Figure 1, is still on the whiteboard.*

HAZEL *and* CODY *standing at the coffee machine, not saying anything.* THEO *enters the scene.*

SCENE I. MATCHING

THEO Good morning everybody! How about looking into efficient matching of regular expressions today?

HAZEL Ok. Can't we use backtracking? What I mean is that we read the given word from left to right, check at the same time whether it matches the given expression, revising decisions when we are not successful. I think this is what algorithms for Perl style regular expressions typically do.

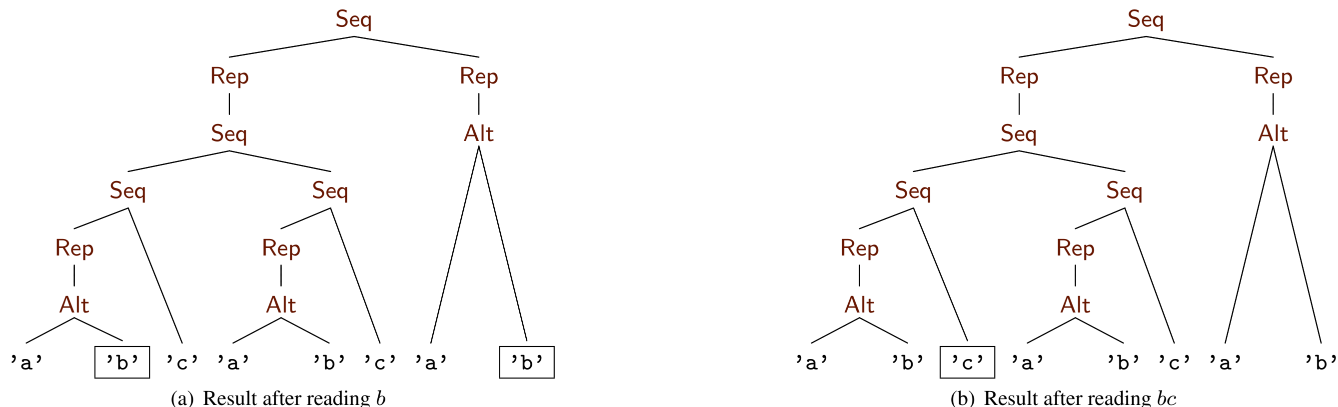
CODY But backtracking is not efficient—at least not always. There are cases where backtracking takes exponential time.

HAZEL Can you give an example?

CODY If you match the word a^n against the regular expression $(a|\varepsilon)^n a^n$, then a backtracking algorithm takes exponential time to find a matching.[1]

HAZEL You're right. When trying to match a^n against $(a|\varepsilon)^n a^n$ one can choose either a or ε in n positions, so all together there are 2^n different options, but only one of them—picking ε every time—leads to a successful matching. A backtracking

[1] By x^n Cody means a sequence of n copies of x.



(a) Result after reading b

(b) Result after reading bc

Figure 2. Marking positions in the regular expression $((a|b)^*c(a|b)^*c)^*(a|b)^*$ while matching

algorithm may pick this combination only after having tried all the other options.— Can we do better?

THEO An alternative is to turn a regular expression into an equivalent deterministic finite-state automaton. The run time for simulating an automaton on a given word is linear in the length of the word, but the automaton can have exponential size in the size of the given regular expression.

CODY That's not good, because then the algorithm not only has exponential space requirements but additionally preprocessing takes exponential time.

HAZEL Can you give an example where the deterministic automaton has necessarily exponential size?

THEO Suppose we are working with the alphabet that contains only a and b. If you want to check whether a word contains two occurrences of a with exactly n characters in between, then any deterministic automaton for this will have 2^{n+1} different states.

CODY Why?

THEO Because at any time—while reading a word from left to right—the automaton needs to know for each of the previous n characters whether it was an a or not in order to tell whether the entire word is accepted.

HAZEL I see. You need such detailed information because if there was an a exactly $n + 1$ positions before the current position and the next character is not an a, then you have to go to the next state, where you will need to know whether there was an a exactly n positions before the current position, and so on.

THEO Exactly!— And here is a formal proof. Suppose an automaton had less than 2^{n+1} states. Then there would be two distinct words of length $n + 1$, say u and v, which, when read by the automaton, would lead to the same state. Since u and v are distinct, there is some position i such that u and v differ at position i, say u carries symbol a in this position, but v doesn't. Now, consider the word w which starts with i copies of b, followed by one occurrence of a ($w = b^i a$). On the one hand, uw has the above property, namely two occurrences of a's with n characters in between, but vw has not, on the other hand, the automaton would get to the same state for both words, so either both words are accepted, or none of them is—a contradiction.

CODY Interesting. And, indeed, a regular expression to solve this task has size only linear in n. If we restrict ourselves to the alphabet consisting of a and b, then we can write it as follows. (*Grabs a pen from his pocket and a business card from his wallet. Scribbles on the back of the business card. Reads aloud the following term.*)

$$(a|b)^*a(a|b)^n a(a|b)^*$$

HAZEL Can we avoid constructing the automaton in advance?

CODY Instead of generating all states in advance, we can generate the initial state and generate subsequent states on the fly. If we discard previous states, then the space requirements are bound by the space requirements of a single state.

THEO And the run time for matching a word of length n is in $O(mn)$ if it takes time $O(m)$ to compute a new state from the previous one.

HAZEL That sounds reasonably efficient. How can we implement this idea?

THEO Glushkov proposed a nice idea for constructing non-deterministic automata from regular expressions, which may come in handy here. It avoids ε-transitions and can probably be implemented in a structural fashion.

HAZEL (*smiling*) I think we would say it could be implemented in a purely functional way.

CODY (*getting excited*) How are his automata constructed?

THEO A state of a Glushkov automaton is a position of the given regular expression where a position is defined to be a place where a symbol occurs. What we want to do is determinize this automaton right away, so we should think of a state as a set of positions.

HAZEL What would such a set mean?

THEO The positions contained in a state describe where one would get to by trying to match a word in every possible way.

HAZEL I don't understand. Can you give an example?

THEO Instead of writing sets of positions, I mark the symbols in the regular expression with a box. This is more intuitive and allows me to explain how a new set of positions is computed from an old one.

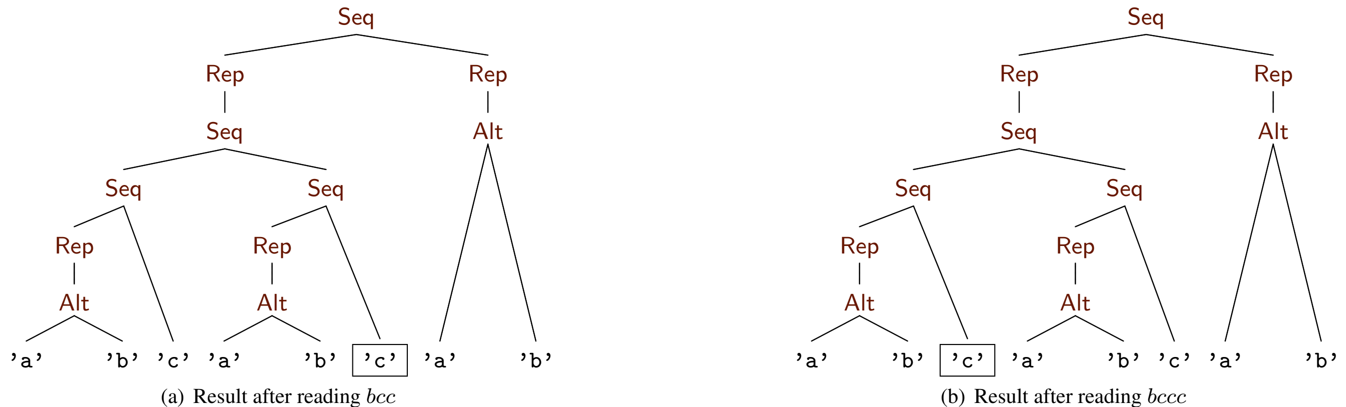
CODY Let's match the string bc against the regular expression which checks whether it contains an even number of c's.

THEO Well, I need to draw some pictures.

CODY Then let's go back to Hazel's office; we can probably use what we wrote on the whiteboard yesterday.

The three move to the left side of the stage, get in front of the whiteboard, where Figure 1 is still shown.

THEO Initially, no symbol is marked, i. e., the initial state is the empty set. We then mark every occurrence of `'b'` in the regular expression that might be responsible for reading the first char-



(a) Result after reading *bcc*

(b) Result after reading *bccc*

Figure 3. Shifting symbol marks in repetitions of the regular expression $((a|b)^*c(a|b)^*c)^*(a|b)^*$

acter b of the word. (*Draws two boxes around the first and last* **'b'** *in the regular expression tree, see Figure 2(a).*)

HAZEL There are two possibilities to read the first character b, which correspond to the two positions that you marked. The last **'b'** in the regular expression can be marked because it follows a repetition which accepts the empty word. But the **'b'** in the middle cannot be responsible for matching the first character because it follows the first **'c'** which has not yet been matched.

THEO Exactly! If we now read the next character c we shift the mark from the first **'b'** to the subsequent **'c'**. (*Does so, which leads to Figure 2(b).*)

CODY And the mark of the other **'b'** is discarded because there is no possibility to shift it to a subsequent **'c'**.

THEO Right, you got the idea.— We have reached a final state if there is a mark on a final character.

CODY When is a character final?

THEO When no other character has to be read in order to match the whole regular expression, i.e., if the remaining regular expression accepts the empty word.

HAZEL I think we can elegantly implement this idea in Haskell! Instead of using sets of positions we can represent states as regular expressions with marks on symbols—just as you did on the whiteboard.
(*They move to the desk,* CODY *sits down right in front of the keyboard,* HAZEL *and* THEO *take the two other office chairs.*)

CODY Ok, let's change the data type. We first consider the simple version without semirings. (*Opens the file from the previous scene, adds a data type for regular expressions with marked symbols. They use this file for the rest of the scene.*)

```
data REG = EPS
         | SYM Bool Char
         | ALT  REG REG
         | SEQ  REG REG
         | REP  REG
```

HAZEL Let's implement the shift function. We probably want it to take a possibly marked regular expression and a character to be read and to produce a possibly marked regular expression.

THEO That's not enough. In the beginning, no position is marked. So if we just shift, we'll never mark a position. A similar problem occurs for subterms. If, in the left subexpression of a sequence, a final position is marked, we want this to be taken into account in the right subexpression.

HAZEL We need a third parameter, m, which represents an additional mark that can be fed into the expression. So the arguments of shift are the mark, a possibly marked regular expression, and the character to be read:

```
shift :: Bool → REG → Char → REG
```

The result of shift is a new regular expression with marks.

CODY The rule for ε is easy, because ε doesn't get a mark:

```
shift _ EPS _ = EPS
```

THEO And a symbol in the new expression is marked if, first, some previous symbol was marked, indicated by m == True, and, second, the symbol equals the character to be read.

```
shift m (SYM _ x) c = SYM (m ∧ x == c) x
```

HAZEL We treat both arguments of a choice of regular expressions the same.

```
shift m (ALT p q) c = ALT (shift m p c) (shift m q c)
```

Sequences are trickier. The given mark is shifted to the first part, but we also have to shift it to the second part if the first part accepts the empty word. Additionally, if the first part contains a final character we have to shift its mark into the second part, too. Assuming helper functions empty and final which check whether a regular expression accepts the empty word or contains a final character, respectively, we can handle sequences as follows:

```
shift m (SEQ p q) c =
  SEQ (shift m p c)
      (shift (m ∧ empty p ∨ final p) q c)
```

We haven't talked about repetitions yet. How do we handle them?

THEO Let's go back to the example first. Assume we have already read the word bc but now want to read two additional c's. (*Stands up, grabs a pen, and changes the tree on the whiteboard; the result is shown in Figure 3(a).*) After reading the first c the mark is shifted from the first **'c'** in the regular expression to the second **'c'** as usual because the repetition in between accepts the empty word. But when reading the second c, the mark is shifted from the second **'c'** in the expression back to the first one! (*Modifies the drawing again, see Figure 3(b) for the result.*) Repetitions can be read multiple times and thus marks have to be passed through them multiple times, too.

CODY So we can complete the definition of shift as follows:

```
shift m (REP r) c = REP (shift (m ∨ final r) r c)
```

We shift a mark into the inner expression if a previous character was marked or a final character in the expression is marked.

HAZEL Ok, let's define the helper functions empty and final. I guess, this is pretty straightforward. (*Types the definition of* empty *in her text editor.*)

```
empty :: REG → Bool
empty EPS         = True
empty (SYM _ _)   = False
empty (ALT p q)   = empty p ∨ empty q
empty (SEQ p q)   = empty p ∧ empty q
empty (REP r)     = True
```

No surprises here. How about final? (*Goes on typing.*)

```
final :: REG → Bool
final EPS         = False
final (SYM b _)   = b
final (ALT p q)   = final p ∨ final q
final (SEQ p q)   = final p ∧ empty q ∨ final q
final (REP r)     = final r
```

CODY (*pointing to the screen*) The case for sequences is wrong. It looks similar to the definition in shift, but you mixed up the variables p and q. (*Takes the keyboard and wants to change the definition.*)

HAZEL No, stop! This is correct. final analyzes the regular expression in the other direction. A final character of the first part is also a final character of the whole sequence if the second part accepts the empty word. Of course, a final character in the second part is always a final character of the whole sequence, as well.

CODY Got it. Let's wrap all this up into an efficient function match for regular expression matching. (*Continues typing.*) The type of match is the same as the type of accept—our previously defined specification.

```
match :: REG → String → Bool
```

If the given word is empty, we can check whether the expression matches the empty word using empty:

```
match r [] = empty r
```

If the given word is a nonempty word c : cs we mark all symbols of the given expression which may be responsible for matching the first character c by calling shift True r c. Then we subsequently shift the other characters using shift False.

```
match r (c : cs) =
  final (foldl (shift False) (shift True r c) cs)
```

THEO Why has the argument to be False?

CODY Because, after having processed the first character, we only want to shift existing marks without adding new marks from the left. Finally, we check whether the expression contains a final character after processing the whole input word.

HAZEL That is a pretty concise implementation of regular expression matching! However, I'm not yet happy with the definition of shift and how it repeatedly calls the auxiliary functions empty and final which traverse their argument in addition to the traversal by shift. Look at the rule for sequences again! (*Points at the* shift *rule for sequences on the screen.*)

```
shift m (SEQ p q) c =
  SEQ (shift m p c)
      (shift (m ∧ empty p ∨ final p) q c)
```

```
data REG_w c s = REG_w {empty_w :: s,
                        final_w :: s,
                        reg_w   :: RE_w c s}

data RE_w c s = EPS_w
              | SYM_w (c → s)
              | ALT_w (REG_w c s) (REG_w c s)
              | SEQ_w (REG_w c s) (REG_w c s)
              | REP_w (REG_w c s)

eps_w :: Semiring s ⇒ REG_w c s
eps_w = REG_w {empty_w = one,
               final_w = zero,
               reg_w   = EPS_w}

sym_w :: Semiring s ⇒ (c → s) → REG_w c s
sym_w f = REG_w {empty_w = zero,
                 final_w = zero,
                 reg_w   = SYM_w f}

alt_w :: Semiring s ⇒ REG_w c s → REG_w c s → REG_w c s
alt_w p q = REG_w {empty_w = empty_w p ⊕ empty_w q,
                   final_w = final_w p ⊕ final_w q,
                   reg_w   = ALT_w p q}

seq_w :: Semiring s ⇒ REG_w c s → REG_w c s → REG_w c s
seq_w p q =
  REG_w {empty_w = empty_w p ⊗ empty_w q,
         final_w = final_w p ⊗ empty_w q ⊕ final_w q,
         reg_w   = SEQ_w p q}

rep_w :: Semiring s ⇒ REG_w c s → REG_w c s
rep_w r = REG_w { empty_w = one,
                  final_w = final_w r,
                  reg_w   = REP_w r}

match_w :: Semiring s ⇒ REG_w c s → [c] → s
match_w r []       = empty_w r
match_w r (c : cs) =
  final_w (foldl (shift_w zero · reg_w) (shift_w one (reg_w r) c) cs)

shift_w :: Semiring s ⇒ s → RE_w c s → c → REG_w c s
shift_w _ EPS_w       _ = eps_w
shift_w m (SYM_w f)   c = (sym_w f) {final_w = m ⊗ f c}
shift_w m (ALT_w p q) c =
  alt_w (shift_w m (reg_w p) c) (shift_w m (reg_w q) c)
shift_w m (SEQ_w p q) c =
  seq_w (shift_w m (reg_w p) c)
        (shift_w (m ⊗ empty_w p ⊕ final_w p) (reg_w q) c)
shift_w m (REP_w r)   c =
  rep_w (shift_w (m ⊕ final_w r) (reg_w r) c)
```

Figure 4. Efficient matching of weighted regular expressions

There are three calls which traverse p and one of them is a recursive call to shift. So, if p contains another sequence where the left part contains another sequence where the left part contains another sequence and so on, this may lead to quadratic run time in the size of the regular expression. We should come up with implementations of empty and final with constant run time.

CODY We need to cache the results of empty and final in the inner nodes of regular expressions such that we don't need to recompute them over and over again. Then the run time of shift is linear in the size of the regular expression and the run time of match is in $O(mn)$ if m is the size of the regular expression and n the length of the given word.

THEO That's interesting. The run time is independent of the number of transitions in the corresponding Glushkov automaton. The reason is that we use the structure of the regular expression to determine the next state and find it without considering all possible transitions.

HAZEL And the memory requirements are in $O(m)$, because we discard old states while processing the input.

The three are excited. THEO *sits down again to join* CODY *and* HAZEL *for generalizing the implementation previously developed: they use semirings again and implement the idea of caching the results of* empty *and* final. *The result is presented in Figure 4.*

HAZEL First of all, we have to add fields to our regular expressions in which we can store the cached values for empty and final. This can be done elegantly by two alternating data types $\mathsf{REG_w}$ and $\mathsf{RE_w}$.

THEO Okay, but what are the curly brackets good for.

CODY This is Haskell's notation for specifying field labels. You can read this definition as a usual data type definition with a single constructor $\mathsf{REG_w}$ taking three arguments. Furthermore, Haskell automatically defines functions $\mathsf{empty_w}$, $\mathsf{final_w}$, and $\mathsf{reg_w}$ to select the values of the corresponding fields.

THEO I understand. How can we go on then?

CODY We use smart constructors, which propagate the cached values automatically. When shifting marks, which are now weights, through the regular expression these smart constructors will come in handy ($\mathsf{eps_w}$, $\mathsf{sym_w}$, $\mathsf{alt_w}$, $\mathsf{seq_w}$, and $\mathsf{rep_w}$).

THEO And again the curly brackets are Haskell's syntax for constructing records?

HAZEL Right. Finally, we have to modify match and shift such that they use cached values and construct the resulting regular expression by means of the smart constructors. The rule for $\mathsf{SYM_w}$ introduces a new final value and caches it as well.

THEO And here the curly brackets are used to update an existing record in only one field.— Two more weeks of Haskell programming with you guys, and I will be able to write beautiful Haskell programs.

Disbelief on HAZEL*'s and* CODY*'s faces.— They all laugh.*

SCENE II. HEAVY WEIGHTS

HAZEL, CODY, *and* THEO *sitting relaxed on office chairs, facing the audience, holding coffee mugs.*

CODY I have a question. Until now we compute a weight for a whole word according to a regular expression. Usually, one is interested in matching a subword of a given word and finding out information about the part that matches.

HAZEL Like the position or the length of the matching part.

CODY Can we use our approach to match only parts of a word and compute information about the matching part?

THEO I think so. How about putting something like $(a|b)^*$ around the given regular expression? This matches anything before and after a part in the middle that matches the given expression.

HAZEL Yes, that should work. And we can probably zip the input list with a list of numbers in order to compute information about positions and lengths. Let's see. (*Scratches her head with one hand and lays her chin into the other.*)

After a few seconds, the three move with their office chairs to the desk again, open the file from before in an editor, and continue typing.

```
submatch_w :: Semiring s ⇒ REG_w (Int, c) s → [c] → s
submatch_w r s =
  match_w (seq_w arb (seq_w r arb)) (zip [0 . .] s)
  where arb = rep_w (sym_w (λ_ → one))
```

CODY I see! arb is a regular expression that matches an arbitrary word and always yields the weight one. And r is a regular expression where the symbols can access information about the position of the matched symbol.

HAZEL Exactly!

THEO How can we create symbols that use the positional information?

HAZEL For example, by using a subclass of Semiring with an additional operation to compute an element from an index:

```
class Semiring s ⇒ Semiring_i s where
  index :: Int → s
```

CODY We can use it to define a variant of the sym function:

```
sym_i :: Semiring_i s ⇒ Char → REG_w (Int, Char) s
sym_i c = sym_w weight
  where weight (pos, x) | x == c    = index pos
                        | otherwise = zero
```

THEO Ok, it yields zero if the character does not match, just like before, but uses the index function to compute a weight from the position of a character. And if we use $\mathsf{sym_i}$ we get regular expressions of type $\mathsf{Reg_w}$ (Int, Char) s, which we can pass to $\mathsf{submatch_w}$. Now, we need some instances of $\mathsf{Semiring_i}$ that use this machinery!

HAZEL How about computing the starting position for a nonempty leftmost subword matching? We can use the following data types:

```
data Leftmost = NoLeft  | Leftmost Start
data Start    = NoStart | Start Int
```

NoLeft is the zero of the semiring, i.e., it represents a failing match. Leftmost NoStart is the one and, thus, used for ignored characters:

```
instance Semiring Leftmost where
  zero = NoLeft
  one  = Leftmost NoStart
```

CODY Let me try to define addition. NoLeft is the identity for ⊕. So the only interesting case is when both arguments are constructed by Leftmost.

```
NoLeft     ⊕ x          = x
x          ⊕ NoLeft     = x
Leftmost x ⊕ Leftmost y = Leftmost (leftmost x y)
  where leftmost NoStart   NoStart   = NoStart
        leftmost NoStart   (Start i) = Start i
        leftmost (Start i) NoStart   = Start i
        leftmost (Start i) (Start j) = Start (min i j)
```

The operation ⊕ is called on the results of matching a choice of two alternative regular expressions. Hence, the leftmost function picks the leftmost of two start positions by computing their minimum. NoStart is an identity for leftmost.

HAZEL Multiplication combines different results from matching the parts of a sequence of regular expressions. If one part fails, then the whole sequence does, and if both match, then the start

position is the start position of the first part unless the first part is ignored:

$$\begin{aligned}
&\mathsf{NoLeft} \otimes _ = \mathsf{NoLeft} \\
&_ \otimes \mathsf{NoLeft} = \mathsf{NoLeft} \\
&\mathsf{Leftmost}\ x \otimes \mathsf{Leftmost}\ y = \mathsf{Leftmost}\ (\mathsf{start}\ x\ y) \\
&\quad \mathbf{where}\ \mathsf{start}\ \mathsf{NoStart}\ s = s \\
&\qquad\quad \mathsf{start}\ s\ _ = s
\end{aligned}$$

THEO We need to make Leftmost an instance of $\mathsf{Semiring_i}$. I guess we just wrap the given position in the Start constructor.

$$\begin{aligned}
&\mathbf{instance}\ \mathsf{Semiring_i}\ \mathsf{Leftmost}\ \mathbf{where} \\
&\quad \mathsf{index} = \mathsf{Leftmost} \cdot \mathsf{Start}
\end{aligned}$$

HAZEL Right. Now, executing $\mathsf{submatch_w}$ in the Leftmost semiring yields the start position of the leftmost match. We don't have to write a new algorithm but can use the one that we defined earlier and from which we know it is efficient. Pretty cool.

THEO Let me see if our program works. (*Starts GHCi.*) I'll try to find substrings that match against the regular expression $a(a|b)^*a$ and check where they start.

```
ghci> let a = sym_i 'a'
ghci> let ab = rep_w (a `alt_w` sym_i 'b')
ghci> let aaba = a `seq_w` ab `seq_w` a
ghci> submatch_w aaba "ab" :: Leftmost
NoLeft
ghci> submatch_w aaba "aa" :: Leftmost
Leftmost (Start 0)
ghci> submatch_w aaba "bababa" :: Leftmost
Leftmost (Start 1)
```

Ok. Good. In the first example, there is no matching and we get back NoLeft. In the second example, the whole string matches and we get Leftmost (Start 0). In the last example, there are three matching subwords—`"ababa"` starting at position 1, `"aba"` starting at position 1, and `"aba"` starting at position 3—and we get the leftmost start position.

CODY Can we extend this to compute also the length of leftmost longest matches?

HAZEL Sure, we use a pair of positions for the start and the end of the matched subword.

$$\begin{aligned}
&\mathbf{data}\ \mathsf{LeftLong} = \mathsf{NoLeftLong} \mid \mathsf{LeftLong}\ \mathsf{Range} \\
&\mathbf{data}\ \mathsf{Range} = \mathsf{NoRange} \mid \mathsf{Range}\ \mathsf{Int}\ \mathsf{Int}
\end{aligned}$$

The Semiring instance for LeftLong is very similar to the one we defined for Leftmost. We have to change the definition of addition, namely, where we select from two possible matchings. In the new situation, we pick the longer leftmost match rather than only considering the start position. If the start positions are equal, we also compare the end positions:

First, they only sketch how to implement this.

$$\begin{aligned}
&\ldots \\
&\mathsf{LeftLong}\ x \oplus \mathsf{LeftLong}\ y = \mathsf{LeftLong}\ (\mathsf{leftlong}\ x\ y) \\
&\quad \mathbf{where}\ \mathsf{leftlong}\ \ldots \\
&\qquad \mathsf{leftlong}\ (\mathsf{Range}\ i\ j)\ (\mathsf{Range}\ k\ l) \\
&\qquad\quad \mid i < k \lor i \equiv k \land j \geqslant l = \mathsf{Range}\ i\ j \\
&\qquad\quad \mid \mathsf{otherwise} = \mathsf{Range}\ k\ l
\end{aligned}$$

CODY And when combining two matches sequentially, we pick the start position of the first part and the end position of the second part. Pretty straightforward!

$$\begin{aligned}
&\ldots \\
&\mathsf{LeftLong}\ x \otimes \mathsf{LeftLong}\ y = \mathsf{LeftLong}\ (\mathsf{range}\ x\ y) \\
&\quad \mathbf{where}\ \mathsf{range}\ \ldots \\
&\qquad \mathsf{range}\ (\mathsf{Range}\ i\ _)\ (\mathsf{Range}\ _\ j) = \mathsf{Range}\ i\ j
\end{aligned}$$

THEO We also need to define the index function for the LeftLong type:

$$\begin{aligned}
&\mathbf{instance}\ \mathsf{Semiring_i}\ \mathsf{LeftLong}\ \mathbf{where} \\
&\quad \mathsf{index}\ i = \mathsf{LeftLong}\ (\mathsf{Range}\ i\ i)
\end{aligned}$$

And again, we can use the same algorithm that we have used before.

The light fades, the three keep typing, the only light emerges from the screen. After a few seconds, the light goes on again, the sketched Semiring *instance is on the screen.*

HAZEL Let's try the examples from before, but let's now check for leftmost longest matching.

```
ghci> submatch_w aaba "ab" :: LeftLong
NoLeftLong
ghci> submatch_w aaba "aa" :: LeftLong
LeftLong (Range 0 1)
ghci> submatch_w aaba "bababa" :: LeftLong
LeftLong (Range 1 5)
```

The three lean back in their office chairs, sip their coffee, and look satisfied.

SCENE III. EXPERIMENTS

CODY *and* HAZEL *sit in front of the computer screen. It's dark by now, no daylight anymore.*

CODY Before we call it a day, let's check how fast our algorithm is. We could compare it to the `grep` command and use the regular expressions we have discussed so far. (*Opens a new terminal window and starts typing.*)

```
bash> for i in `seq 1 10`; do echo -n a; done | \
 ...> grep -cE "^(a?){10}a{10}$"
1
```

HAZEL What was that?

CODY That was a for loop printing ten a's in sequence which were piped to the `grep` command to print the number of lines matching the regular expression $(a|\varepsilon)^{10}a^{10}$.

HAZEL Aha. Can we run some more examples?

CODY Sure. (*Types in more commands.*)

```
bash> for i in `seq 1 9`; do echo -n a; done | \
 ...> grep -cE "^(a?){10}a{10}$"
0
bash> for i in `seq 1 20`; do echo -n a; done | \
 ...> grep -cE "^(a?){10}a{10}$"
1
bash> for i in `seq 1 21`; do echo -n a; done | \
 ...> grep -cE "^(a?){10}a{10}$"
0
```

HAZEL Ah. You were trying whether nine a's are accepted—they are not—and then checked 20 and 21 a's.

CODY Yes, it seems to work correctly. Let's try bigger numbers and use the `time` command to check how long it takes.

```
bash> time for i in `seq 1 500`;do echo -n a;done |\
 ...> grep -cE "^(a?){500}a{500}$"
```

CODY *and* HAZEL *stare at the screen, waiting for the call to finish. A couple of seconds later it does.*

```
1

real    0m17.235s
user    0m17.094s
sys     0m0.059s
```

HAZEL That's not too fast, is it? Let's try our implementation. (*Switches to GHCi and starts typing.*)

```
ghci> let a = sym_w ('a'==)
ghci> let seq_n n = foldr1 seq_w . replicate n
ghci> let re n = seq_n n (alt_w a eps_w) `seq_w` seq_n n a
ghci> :set +s
ghci> match_w (re 500) (replicate 500 'a')
True
(5.99 secs, 491976576 bytes)
```

CODY Good. We're faster than `grep` and we didn't even compile! But it's using a lot of memory. Let me see. (*Writes a small program to match the standard input stream against the above expression and compiles it using GHC.*) I'll pass the `-s` option to the run-time system so we can see both run time and memory usage without using the `time` command.

```
bash> for i in `seq 1 500`; do echo -n a; done | \
 ...> ./re500 +RTS -s
match
...
  1 MB total memory in use
...
  Total time   0.06s  (0.21s elapsed)
...
```

Seems like we need a more serious competitor!

HAZEL I told you about Google's new library. They implemented an algorithm in C++ with similar worst case performance as our algorithm. Do you know any C++?

CODY Gosh!

> *The light fades, the two keep typing, the only light emerges from the screen. After a few seconds, the light goes on again.*

HAZEL Now it compiles!

CODY Puuh. This took forever—one hour.

HAZEL Let's see whether it works.

CODY C++ isn't Haskell.

> *They both smile.*

HAZEL We wrote the program such that the whole string is matched, so we don't need to provide the start and end markers `^` and `$`.

```
bash> time for i in `seq 1 500`;do echo -n a;done |\
 ...> ./re2 "(a?){500}a{500}"
match
real    0m0.092s
user    0m0.076s
sys     0m0.022s
```

Ah, that's pretty fast, too. Let's push it to the limit:

```
bash> time for i in `seq 1 5000`;do echo -n a;done |\
 ...> ./re2 "(a?){5000}a{5000}"
Error ... invalid repetition size: {5000}
```

CODY Google doesn't want us to check this example. But wait. (*Furrows his brow.*) Let's cheat:

```
bash> time for i in `seq 1 5000`;do echo -n a;done |\
 ...> ./re2 "((a?){50}){100}(a{50}){100}"
match
real    0m4.919s
user    0m4.505s
sys     0m0.062s
```

HAZEL Nice trick! Let's try our program. Unfortunately, we have to recompile for $n = 5000$, because we cannot parse regular expressions from strings yet.

> *They recompile their program and run it on 5000 a's.*

```
bash> for i in `seq 1 5000`; do echo -n a; done |\
 ...> ./re5000 +RTS -s
match
...
  3 MB total memory in use
...
  Total time    20.80s  (21.19s elapsed)
  %GC time      83.4%   (82.6% elapsed)
...
```

HAZEL The memory requirements are quite good but in total it's about five times slower than Google's library in this example.

CODY Yes, but look at the `GC` line! More than 80% of the run time is spent during garbage collection. That's certainly because we rebuild the marked regular expression in each step by shift_w.

HAZEL This seems inherent to our algorithm. It's written as a purely functional program and does not mutate one marked regular expression but computes new ones in each step. Unless we can somehow eliminate the data constructors of the regular expression, I don't see how we can improve on this.

CODY A new mark of a marked expression is computed in a tricky way from multiple marks of the old expression. I don't see how to eliminate the expression structure which guides the computation of the marks.

HAZEL Ok, how about trying another example? The Google library is based on simulating an automaton just like our algorithm. Our second example, which checks whether there are two a's with a specific distance, is a tough nut to crack for automata-based approaches, because the automaton is exponentially large.

> THEO *curiously enters the scene.*

CODY Ok, can we generate an input string such that almost all states of this automaton are reached? Then, hopefully, caching strategies will not be successful.

THEO If we just generate a random string of a's and b's, then the probability that it matches quite early is fairly high. Note that the probability that it matches after $n + 2$ positions is one fourth. We need to generate a string that does not match at all and is sufficiently random to generate an exponential number of different states. If we want to avoid that there are two a's with n characters in between, we can generate a random string and additionally keep track of the $n + 1$ previous characters. Whenever we are exactly $n + 1$ steps after an a, we generate a b. Otherwise, we randomly generate either an a or a b. Maybe, we should ...

> THEO*'s voice fades out. Apparently, he immerses himself in some problem.* CODY *and* HAZEL *stare at him for a few seconds, then turn to the laptop and write a program* `genrnd` *which produces random strings of a's and b's. They turn to* THEO.

CODY Theo, we're done!

THEO Ohhh, sorry! (*Looks at the screen.*)

CODY We can call `genrnd` with two parameters like this:

```
bash> ./genrnd 5 6
bbbaaaaabbbbbbbabaabbbbbbbaabbbbbbabbaabbb
```

The result is a string of a's and b's such that there are no two a's with 5 characters in between. The total number of generated characters is the product of the incremented arguments, i.e., in this case $(5 + 1) * (6 + 1) = 42$.

THEO Ok. So if we want to check our regular expression for $n = 20$ we need to use a string with length greater than $2^{20} \approx 10^6$. Let's generate around 2 million characters.

HAZEL Ok, let's check out the Google program.

```
bash> time ./genrnd 20 100000 | ./re2 ".*a.{20}a.*"
```

While the program is running CODY *is looking at a process monitor and sees that Google's program uses around 5 MB of memory.*

```
no match
real    0m4.430s
user    0m4.514s
sys     0m0.025s
```

Let's see whether we can beat this. First, we need to compile a corresponding program that uses our algorithm.

They write a Haskell program `dist20` *which matches the standard input stream against the regular expression* `.*a.{20}a.*`. *Then they run it.*

```
bash> ./genrnd 20 100000 | ./dist20 +RTS -s
no match
...
  2 MB total memory in use
...
  Total time     3.10s  (3.17s elapsed)
  %GC time       5.8%   (6.3% elapsed)
...
```

HAZEL Wow! This time we are faster than Google. And our program uses only little memory.

CODY Yes, and in this example, the time for garbage collection is only about 5%. I guess that's because the regular expression is much smaller now, so fewer constructors become garbage.

THEO This is quite pleasing. We have not invested any thoughts in efficiency—at least w.r.t. constant factors—but, still, our small Haskell program can compete with Google's library.

HAZEL What other libraries are there for regular expression matching? Obviously, we cannot use a library that performs backtracking, because it would run forever on our first benchmark. Also, we cannot use a library that constructs a complete automaton in advance, because it would eat all our memory in the second benchmark. What does the standard C library do?

CODY No idea.

Just as above, the light fades out, the screen being the only source of light. CODY *and* HAZEL *keep working,* THEO *falls asleep on his chair. After a while, the sun rises.* CODY *and* HAZEL *look tired, they wake up* THEO.

CODY (*addressing* THEO) We wrote a program that uses the standard C library `regex` for regular expression matching and checked it with the previous examples. It's interesting, the performance differs hugely on different computers. It seems that different operating systems come with different implementations of the `regex` library. On this laptop—an Intel MacBook running OS X 10.5—the `regex` library outperforms Google's library in the first benchmark and the Haskell program in the second benchmark – both by a factor between two and three, but not more. We tried it on some other systems, but the library was slower there. Also, when not using the option `RE2::Latin1` in the `re2` program it runs in UTF-8 mode and is more than three times slower in the second benchmark.

THEO Aha.

ACT III

SCENE I. INFINITE REGULAR EXPRESSIONS

HAZEL *sitting at her desk.* THEO *and* CODY *at the coffee machine, eating a sandwich.*

CODY The benchmarks are quite encouraging and I like how elegant the implementation is.

THEO I like our work as well, although it is always difficult to work with practitioners. (*Rolls his eyes.*) It is a pity that the approach only works for regular languages.

CODY I think this is not true. Haskell is a lazy language. So I think there is no reason why we should not be able to work with non-regular languages.

THEO How is this possible? (*Starts eating much faster.*)

CODY Well, I think we could define an infinite regular expression for a given context-free language. There is no reason why our algorithm should evaluate unused parts of regular expressions. Hence, context-free languages should work as well.

THEO That's interesting. (*Finishes his sandwich.*) Let's go to Hazel and discuss it with her.

THEO *jumps up and rushes to* HAZEL. CODY *is puzzled and follows, eating while walking.*

THEO (*addressing* HAZEL) Cody told me that it would also be possible to match context-free languages with our Haskell program. Is that possible?

HAZEL It might be. Let's check how we could define a regular expression for any number of a's followed by the same number of b's ($\{a^n b^n \mid n \geqslant 0\}$).

CODY Instead of using repetitions like in $a^* b^*$, we have to use recursion to define an infinite regular expression. Let's try.

```
ghci> let a = sym_w ('a'==)
ghci> let b = sym_w ('b'==)
ghci> let anbn = eps_w `alt_w` seq_w a (anbn `seq_w` b)
ghci> match_w anbn ""
```

The program doesn't terminate.

```
^C
```

THEO It doesn't work. That's what I thought!

HAZEL You shouldn't be so pessimistic. Let's find out why the program evaluates the infinite regular expression.

CODY I think the problem is the computation of final_w. It traverses the whole regular expression while searching for marks it can propagate further on. Is this really necessary?

HAZEL You mean there are parts of the regular expression which do not contain any marks. Traversing these parts is often superfluous because nothing is changed anyway, but our algorithm currently evaluates the whole regular expression even if there are no marks.

CODY We could add a flag at the root of each subexpression indicating that the respective subexpression does not contain any mark at all. This could also improve the performance in the finite case, since subexpressions without marks can be shared instead of copied by the shift_w function.

THEO I'd prefer to use the term weights when talking about the semiring implementation. When you say marks you mean weights that are non-zero.

CODY Right. Let me change the implementation.

CODY *leaves.*

SCENE II. LAZINESS

THEO *and* HAZEL *still at the desk.* CODY *returns.*

CODY (*smiling*) Hi guys, it works. I had to make some modifications in the code, but it's still a slick program. You can check out the new version now.

HAZEL What did you do?

CODY First of all, I added a boolean field active to the data type REG_w. This field should be False for a regular expression without non-zero weights. If a weight is shifted into a subexpression the corresponding node is marked as active.

```
shift_w m (SYM_w f) c =
  let fin = m ⊗ f c
  in  (sym_w f) {active = fin ≠ zero, final_w = fin}
```

HAZEL So the new field is a flag that tells whether there are any non-zero weights in a marked regular expression. We need an extra flag because we cannot deduce this information from the values of empty_w and final_w alone. An expression might contain non-zero weights even if the value of final_w is zero.

CODY Right. The smart constructors propagate the flag as one would expect. Here is the modified definition of seq_w, the other smart constructors need to be modified in a similar fashion:

```
seq_w :: Semiring s ⇒ REG_w c s → REG_w c s → REG_w c s
seq_w p q =
  REG_w {active  = active p ∨ active q,
         empty_w = empty_w p ⊗ empty_w q,
         final_w = final_a p ⊗ empty_w q ⊕ final_a q,
         reg_w   = SEQ_w p q}
```

HAZEL What is final_a?

CODY It's an alternative to final_w that takes the active flag into account.

```
final_a :: Semiring s ⇒ REG_w c s → s
final_a r = if active r then final_w r else zero
```

HAZEL How does this work?

CODY It blocks the recursive computation of final_w for inactive regular expressions. This works because of lazy evaluation: if the given expression is inactive, this means that it does not contain any non-zero weights. Thus, we know that the result of computing the value for final_w is zero. But instead of computing this zero recursively by traversing the descendants, we just set it to zero and ignore the descendants.

HAZEL This only works if the value of the active field can be accessed without traversing the whole expression. This is why we need special constructors for constructing an initial regular expression with all weights set to zero.

CODY Yes, for example, a constructor function for sequences without non-zero weights can be defined as follows:

```
seq p q = REG_w {active  = False,
                 empty_w = empty_w p ⊗ empty_w q,
                 final_w = zero,
                 reg_w   = SEQ_w p q}
```

The difference to seq_w is in the definition of the active and final_w fields, which are set to False and zero, respectively.

HAZEL Ok, I guess we also need new functions alt and rep for initial regular expressions where all weights are zero.

CODY Right. The last change is to prevent the shift_w function from traversing (and copying) inactive subexpressions. This can be easily implemented by introducing a wrapper function in the definition of shift_w:

```
shift_w :: (Eq s, Semiring s) ⇒
             s → REG_w c s → c → REG_w c s
shift_w m r c | active r ∨ m ≠ zero = step_w m (reg_w r) c
              | otherwise          = r
```

where step_w is the old definition of shift_w with recursive calls to shift_w. The only change is the definition for the SYM_w case, as I showed you before. [2]

THEO Ok, fine (*tired of the practitioners' conversation*). How about trying it out now?

CODY Ok, let's try $a^n b^n$ with the new implementation. We only have to use the variants of our smart constructors that create inactive regular expressions.

```
ghci> let a = sym_w ('a'==)
ghci> let b = sym_w ('b'==)
ghci> let anbn = eps_w `alt` seq a (anbn `seq` b)
ghci> match_w anbn ""
True
ghci> match_w anbn "ab"
True
ghci> match_w anbn "aabb"
True
ghci> match_w anbn "aabbb"
False
```

THEO Impressive. So, what is the class of languages that we can match with this kind of infinite regular expressions?

HAZEL I guess it is possible to define an infinite regular expression for every context-free language. We only have to avoid left recursion.

CODY Right. Every recursive call has to be guarded by a symbol, just as with parser combinators.

THEO I see. Then it is enough if the grammar is in Greibach normal form, i. e., every right-hand side of a rule starts with a symbol.

CODY Exactly. But, in addition, regular operators are allowed as well, just as in extended Backus-Naur form. You can use stars and nested alternatives as well.

HAZEL Pretty cool. And I think we can recognize even more languages. Some context-sensitive languages should work as well.

THEO How should this be possible?

HAZEL By using the power of Haskell computations. It should be possible to construct infinite regular expressions in which each alternative is guarded by a symbol and remaining expressions can be computed by arbitrary Haskell functions. Let's try to specify an infinite regular expression for the language $a^n b^n c^n$ (more precisely, $\{a^n b^n c^n \mid n \geqslant 0\}$), which—as we all know—is not context-free.

THEO A first approach would be something like the following. (*Scribbles on the whiteboard.*)

$$\varepsilon \mid abc \mid aabbcc \mid aaabbbccc \mid \ldots$$

CODY Unfortunately, there are infinitely many alternatives. If we generate them recursively, the recursive calls are not guarded by a symbol. But we can use distributivity of choice and sequence.

HAZEL Ah! Finally, we are using an interesting semiring law:

$$\varepsilon \mid a(bc \mid a(bbcc \mid a(bbbccc \mid a(\ldots))))$$

Now every infinite alternative of a choice is guarded by the symbol a. Hence, our algorithm only traverses the corresponding subexpression if the input contains another a.

CODY So, let's see! We first define functions to generate a given number of b's and c's. (*Typing into GHCi again.*)

```
ghci> let bs n = replicate n (sym_w ('b'==))
ghci> let cs n = replicate n (sym_w ('c'==))
```

Then we use them to build our expression. (*Continues typing.*)

[2] These modifications not only allow infinite regular expressions, they also affect the performance of the benchmarks discussed in Act II. The first benchmark runs in about 60% of the original run time. The run time of the second is roughly 20% worse. Memory usage does not change siginificantly.

```
ghci> let bcs n = foldr1 seq (bs n ++ cs n)
ghci> let a = symw ('a'==)
ghci> let abc n = a `seq` alt (bcs n) (abc (n+1))
ghci> let anbncn = epsw `alt` abc 1
```

THEO Fairly complicated! Can you check it?

CODY enters some examples.

```
ghci> matchw anbncn ""
True
ghci> matchw anbncn "abc"
True
ghci> matchw anbncn "aabbcc"
True
ghci> matchw anbncn "aabbbcc"
False
```

THEO Great, it works. Impressive!

SCENE III. REVIEW

The three at the coffee machine.

HAZEL Good that you told us about Glushkov's construction.

THEO We've worked for quite a long time on the regular expression problem now, but did we get somewhere?

HAZEL Well, we have a cute program, elegant, efficient, concise, solving a relevant problem. What else do you want?

CODY What are we gonna do with it? Is it something people might be interested in?

THEO I find it interesting, but that doesn't count. Why don't we ask external reviewers? Isn't there a conference deadline coming up for nice programs (*smiling*)?

CODY and HAZEL (*together*) ICFP.

THEO ICFP?

CODY Yes, they collect functional pearls—elegant, instructive, and fun essays on functional programming.

THEO But how do we make our story a fun essay?

The three turn to the audience, bright smiles on their faces!

EPILOGUE

Regular expressions were introduced by Stephen C. Kleene in his 1956 paper [Kleene 1956], where he was interested in characterizing the behavior of McCulloch-Pitts nerve (neural) nets and finite automata, see also the seminal paper [Rabin and Scott 1959] by Michael O. Rabin and Dana Scott. Victor M. Glushkov's paper from 1960, [Glushkov 1960], is another early paper where regular expressions are translated into finite-state automata, but there are many more, such as the paper by Robert McNaughton and H. Yamada, [McNaughton and Yamada 1960]. Ken Thompson's paper from 1968 is the first to describe regular expression matching [Thompson 1968].

The idea of introducing weights into finite automata goes back to a paper by Marcel P. Schützenberger, [Schützenberger 1961]; weighted regular expressions came up later. A good reference for the weighted setting is the Handbook of Weighted Automata [Droste et al. 2009]; one of the papers that is concerned with several weighted automata constructions is [Allauzen and Mohri 2006]. The paper [Caron and Flouret 2003] is one of the papers that focuses on Glushkov's construction in the weighted setting.

What we nowadays call Greibach normal form is defined in Sheila A. Greibach's 1965 paper [Greibach 1965].

Haskell is a lazy, purely functional programming language. A historical overview is presented in [Hudak et al. 2007]. There are several implementations of regular expressions in Haskell [Haskell Wiki]. Some of these are bindings to existing C libraries, others are implementations of common algorithms in Haskell. In comparison with these implementations our approach is much more concise and elegant, but can still compete with regard to efficiency. The experiments were carried out using GHC version 6.10.4 with `-O2` optimizations.

The Google library can be found at `http://code.google.com/p/re2/`, the accompanying blog post at `http://google-opensource.blogspot.com/2010/03/re2-principled-approach-to-regular.html`.

REFERENCES

C. Allauzen and M. Mohri. A unified construction of the Glushkov, follow, and Antimirov automata. In R. Kralovic and P. Urzyczyn, editors, *Mathematical Foundations of Computer Science 2006 (MFCS 2006), Stará Lesná, Slovakia*, volume 4162 of *Lecture Notes in Computer Science*, pages 110–121. Springer, 2006.

P. Caron and M. Flouret. From Glushkov WFAs to rational expressions. In Z. Ésik and Z. Fülöp, editors, *Developments in Language Theory, 7th International Conference (DLT 2003), Szeged, Hungary*, volume 2710 of *Lecture Notes in Computer Science*, pages 183–193. Springer, 2003.

M. Droste, W. Kuich, and H. Vogler. *Handbook of Weighted Automata*. Springer, New York, 2009.

V. M. Glushkov. On a synthesis algorithm for abstract automata. *Ukr. Matem. Zhurnal*, 12(2):147–156, 1960.

S. A. Greibach. A new normal-form theorem for context-free phrase structure grammars. *J. ACM*, 12(1):42–52, 1965.

Haskell Wiki. Haskell – regular expressions. `http://www.haskell.org/haskellwiki/Regular_expressions`.

P. Hudak, J. Hughes, S. L. Peyton-Jones, and P. Wadler. A history of Haskell: being lazy with class. In *Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III), San Diego, California*, pages 1–55. ACM, 2007.

S. Kleene. Representation of events in nerve nets and finite automata. In C. Shannon and J. McCarthy, editors, *Automata Studies*, pages 3–42. Princeton University Press, Princeton, N.J., 1956.

R. McNaughton and H. Yamada. Regular expressions and state graphs for automata. *IEEE Transactions on Electronic Computers*, 9(1):39–47, 1960.

M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM journal of research and development*, 3(2):114–125, 1959.

M. P. Schützenberger. On the definition of a family of automata. *Information and Control*, 4(2–3):245–270, 1961.

K. Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, 1968.