Bottom-up rewriting with smart constructors, hereditary substitution & normalization by evaluation

Jules Jacobs

May 6, 2021

Abstract

In this note I explain these three well-known techniques for rewriting to normal form. We'll look at how to use them to optimize a regular expression and compute the β normal form of a lambda term. We will see that these three techniques share the same key idea.

Contents

1 Introduction

Suppose we want to simplify regular expressions consisting of the following operations. The symbol 0 represents the regex that doesn't match anything, 1 represents the regex that only matches the empty string, 'c' represents a single character, $(+)$ represents union, $(\cdot)$ represents concatenation, and $*$ represents repetition:

$$r \in \mathsf{Re} ::= 0 \mid 1 \mid \text{`}c\text{`} \mid r + r \mid r \cdot r \mid r^*,$$

We want to rewrite using the following equations from left to right as much as possible:

$$\begin{array}{lll} r + 0 = r & r \cdot 0 = 0 & 0^* = 1 \\ 0 + r = r & 0 \cdot r = 0 & 1^* = 1 \\ r + r = r & r \cdot 1 = r & (r^*)^* = r^* \\ (r + s) + t = r + (s + t) & 1 \cdot r = r & \\ & (r \cdot s) \cdot t = r \cdot (s \cdot t) & \end{array}$$

For example, $(a \cdot 0^*)^{**} + 0 = a^*$.

Simplifying regexes using those equations is useful for implementing regular expression matching with Brzozowski derivatives [Brz64, ORT09], though the point isn't this particular example; rewriting expressions to normal form for a given set of equations is more broadly useful.

The naive way to do this is to take the regular expression r, and try to find some subnode of r where one of the left hand sides of the equations can be rewritten to the right hand side. If we repeat this as much as possible, until no equation matches any subnode, we have rewritten the regex to normal form. The problem with this approach is that it is extremely inefficient and not even very easy to implement. At each step we have to search through r to find a place to apply a rewrite rule.

A more systematic way to do this is to schedule the rewrites bottom up. For instance, if $r = r_1 + r_2$ then we first recursively rewrite r_1 and r_2 to normal form. We then only need to check if $r_1 + r_2$ itself is in normal form. If it is, then we're done. If one of the left hand sides of the equations match, then we apply the rewrite rule. We then start the whole normalization process all over again, because after we've applied the rewrite rule there might be new opportunities for further rewriting.

This is better, but still not great. Suppose $r = (r_1 + r_2) + r_3$, and we've already rewritten r_1, r_2, r_3 to normal form. Now the associativity rewrite rule wants to rewrite this to $r_1 + (r_2 + r_3)$. This is a new regular expression, so maybe more rewrite rules match. However, we do know that r_1, r_2, r_3 themselves are still in normal form. So in order to rewrite $r_1 + (r_2 + r_3)$ to normal form, we don't need to recursively re-normalize r_1, r_2, r_3. We only need to check if any rewrite rule matches for the two newly created $(+)$ nodes. We thus want to keep track of which nodes are already in normal form, so that we never need to recurse into them again to uselessly try and fail to rewrite them further. Note we do need to *look into* nodes that are already in normal form: if $r_2 = r_{21} + r_{22}$ then further rewrites do apply to $r_2 + r_3$, even if r_2 and r_3 are in normal form. This may seem like it can get a little bit complicated, but in the next section we'll discuss a well known technique to do this easily and very efficiently.

2 Bottom-up rewriting with smart constructors

Let us first define a data type of regular expressions:

```
class Re
object Emp extends Re // 0
object Eps extends Re // 1
case class Chr(a:Char) extends Re // 'c'
case class Seq(a:Re, b:Re) extends Re // r · s
case class Alt(a:Re, b:Re) extends Re // r + s
case class Star(a:Re) extends Re // r*
```

2.1 Smart constructors

The key idea is to define *smart constructors* `seq,alt,star` for the ordinary constructors `Seq,Alt,Star`. We want these smart constructors to satisfy the following property:

If we use a smart constructor on values that are in normal form, it must return a value in normal form.

Here's the one for `seq`:

```
def seq(a:Re, b:Re):Re =
  (a,b) match {
    case (Emp,_)  ⟹  Emp
    case (_,Emp)  ⟹  Emp
    case (Eps,x)  ⟹  x
```

```
    case (x,Eps) ⟹ x
    case (Seq(x,y),b) ⟹ seq(x,seq(y,b))
    case _ ⟹ Seq(a,b)
  }
```

We check if any of the equations for $(\cdot)$ match, and if so we return the right hand side. Whenever we construct a new node after a rewrite, we *have to use the smart constructors*. That guarantees that the returned value is in normal form. If no equation matches (last case), we can use the ordinary constructor `Seq`.

Here are the smart constructors `alt,star`:

```
def alt(a:Re, b:Re):Re =
  (a,b) match {
    case (Emp,x) ⟹ x
    case (x,Emp) ⟹ x
    case (Alt(x,y),b) ⟹ alt(x,alt(y,b))
    case _ ⟹ if(a==b) a else Alt(a,b)
  }

def star(a:Re):Re =
  a match {
    case Emp ⟹ Eps
    case Eps ⟹ Eps
    case Star(_) ⟹ a
    case _ ⟹ Star(a)
  }
```

2.2 Converting to normal form

To convert a regular expression to normal form, we simply *"copy"* it with the smart constructors:

```
def nf(a:Re):Re =
  a match {
    case Emp ⟹ Emp
    case Eps ⟹ Eps
    case Chr(c) ⟹ Chr(c)
    case Alt(a,b) ⟹ alt(nf(a),nf(b))
    case Seq(a,b) ⟹ seq(nf(a),nf(b))
    case Star(a) ⟹ star(nf(a))
  }

val r = Alt(Star(Star(Seq(Chr('a'),Star(Emp)))),Emp)
nf(r) // Star(Chr('a'))
```

By the property that smart constructors return normal forms if you pass them normal forms, this function will return a normal form. What's more, this is very efficient: we only recurse over the initial regular expression *once*, and we *only ever allocate regular expressions that are in normal form*. We never allocate an intermediate value like $(r_1 + r_2) + r_3$ to which a rewrite rule applies; we rewrite it before even constructing it.

2.3 Handling commutativity

Suppose we also want to use commutativity $r + s = s + r$. This is nice, because then if we have $(r + s) + r$ we can use commutativity and associativity to rewrite that to $s + (r + r)$, so that the cancellation rule $r + r = r$ can be used to simplify it. We can't simply keep rewriting from right to

left, because that would result in an infinite loop. What we want is to bring equal regexes next to each other, so that the cancellation rule applies.

To do this, we define an *ordering* ($<$) on regular expressions, and rewrite $r + s = s + r$ only if $s < r$. This will bring longer sequences $r_1 + r_2 + \cdots + r_n$ into sorted order, so that adjacent equal elements can be canceled. Any ordering will do. A convenient option is to sort them by their hash code. That leads to the following smart constructor:

```
def alt1(a:Re, b:Re):Re =
  (a,b) match {
    case (Emp,x) ⟹ x
    case (x,Emp) ⟹ x
    case (Alt(x,y),b) ⟹ alt1(x,alt1(y,b))
    case (a,Alt(x,y)) ⟹
      if(a==x) b
      else if(a.hashCode() < x.hashCode()) Alt(a,b)
      else alt1(x,alt1(a,y))
    case _ ⟹
      if(a==b) a
      else if(a.hashCode() < b.hashCode()) Alt(a,b)
      else Alt(b,a)
  }
```

This smart constructor is able to do that optimization:

```
val a = Chr('a')
val b = Chr('b')
alt(a,alt(b,a)) // Alt(Chr('a'), Alt(Chr('b'), Chr('a')))
alt1(a,alt1(b,a)) // Alt(Chr('b'), Chr('a'))
```

2.4 Optimizing at parse time

As you can see in the previous example, an alternative to first constructing a regex and then converting it to normal form, is to use the smart constructors to construct the initial regex in the first place. The parser could call the smart constructors instead of the ordinary constructors.

This is what the JVM does. It speeds up the JIT compiler because simple local rewrite rules are able to shrink the IR significantly, so the rest of the compiler has to wade though less code. In fact, the local rewrite rules are so effective in combination with the sea of nodes IR, that you could potentially write a reasonably good compiler by just doing optimization with smart constructors [CP95].

3 Better normal form representations

The implementation of the smart constructor that handles commutativity is rather complicated. We're essentially implementing a very bad version of bubble sort. We even need separate cases for an element in the middle of the list and an element at the end of the list.

A better way is to use a representation of regular expressions tailored to normal forms. We represent n-ary sequential composition as a list. This builds associativity $(r \cdot s) \cdot t = r \cdot (s \cdot t)$ into the representation. We represent n-ary alternative with a set. This builds associativity $(r + s) + t = r + (s + t)$

```
class Re2
case class Chr2(a:Char) extends Re2
case class Seq2(rs:List[Re2]) extends Re2
case class Alt2(rs:Set[Re2]) extends Re2
case class Star2(r:Re2) extends Re2
```

```
val emp2 = Alt2(Set())
val eps2 = Seq2(List())

def seq2(rs:List[Re2]):Re2 = {
  val rs2 = rs.flatMap{case Seq2(rs) ⟹ rs
                       case x ⟹ List(x)}
  if(rs2.contains(emp2)) emp2
  else if(rs2.size == 1) rs2.head
  else Seq2(rs2)
}

def alt2(rs:Set[Re2]):Re2 = {
  val rs2 = rs.flatMap{case Alt2(rs) ⟹ rs
                       case x ⟹ Set(x)}
  if(rs2.size == 1) rs2.head
  else Alt2(rs2)
}

def star2(a:Re2):Re2 =
  a match {
    case Alt2(rs) if rs.isEmpty ⟹ eps2
    case Seq2(rs) if rs.isEmpty ⟹ eps2
    case Star2(_) ⟹ a
    case _ ⟹ Star2(a)
  }


// We can define conversion functions from Re to Re2 and vice versa that put the regex in norm
// Alternatively we could always use the Re2 representation

def reToRe2(r:Re):Re2 =
  r match {
    case Eps ⟹ eps2
    case Emp ⟹ emp2
    case Chr(c) ⟹ Chr2(c)
    case Alt(a,b) ⟹ alt2(Set(reToRe2(a),reToRe2(b)))
    case Seq(a,b) ⟹ seq2(List(reToRe2(a),reToRe2(b)))
    case Star(a) ⟹ star2(reToRe2(a))
  }

def fold1[A](xs:Iterable[A], z:A, f:(A,A) ⟹ A):A = {
  if(xs.isEmpty) z
  else {
    var y = xs.head
    for(x <- xs.tail) y = f(x,y)
    return y
  }
}

def re2ToRe(r:Re2):Re =
  r match {
    case Chr2(c) ⟹ Chr(c)
    case Seq2(rs) ⟹ fold1(rs.map(re2ToRe), Eps, Seq)
    case Alt2(rs) ⟹ fold1(rs.map(re2ToRe), Eps, Alt)
    case Star2(r) ⟹ Star(re2ToRe(r))
  }

val a = Chr2('a')
val b = Chr2('b')
```

```
val z = alt2(Set(a,b,emp2,eps2))
alt2(Set(z,z,a))
seq2(List(emp2, a, b))
re2ToRe(z)
```

4 Hereditary substitution

[KA10]

$$e \in \mathsf{Tm} ::= x \mid \lambda x.e \mid \mathsf{app}(e,e)$$

$$\mathsf{app}((\lambda x.e_1), e_2) \rightsquigarrow e_1[x := e_2]$$

```
// Normalization by hereditary substitution for De Bruijn terms

class Tm
case class Var(n:Int) extends Tm
case class Lam(a:Tm) extends Tm
case class App(a:Tm,b:Tm) extends Tm

// Renaming

def liftR(f : Int ⟹ Int): Int ⟹ Int =
  (n) ⟹ if(n==0) 0 else f(n-1) + 1

def rename(a:Tm, f:Int ⟹ Int):Tm =
  a match {
    case Var(n) ⟹ Var(f(n))
    case Lam(a) ⟹ Lam(rename(a,liftR(f)))
    case App(a,b) ⟹ App(rename(a,f),rename(b,f))
  }

// Substitution

def shift(e:Tm, f:Int ⟹ Tm):Int ⟹ Tm =
  (n) ⟹ if(n==0) e else f(n-1)

def liftS(f : Int ⟹ Tm):Int ⟹ Tm =
  shift(Var(0), k ⟹ rename(f(k), (_+1)))

def subst(a:Tm, f:Int ⟹ Tm):Tm =
  a match {
    case Var(n) ⟹ f(n)
    case Lam(a) ⟹ Lam(subst(a,liftS(f)))
    case App(a,b) ⟹ App(subst(a,f),subst(b,f))
  }

def subst1(a:Tm, b:Tm):Tm = subst(a, shift(b,Var))

// Hereditary substitution

def app(a:Tm, b:Tm):Tm =
```

```
  a match {
    case Lam(e) ⟹ subst1(e, b)
    case _ ⟹ App(a,b)
  }

def hsubst(a:Tm, f:Int ⟹ Tm):Tm =
  a match {
    case Var(n) ⟹ f(n)
    case Lam(a) ⟹ Lam(hsubst(a,liftS(f)))
    case App(a,b) ⟹ app(hsubst(a,f),hsubst(b,f))
  }

def hsubst1(a:Tm, b:Tm):Tm = hsubst(a, shift(b,Var))

def norm(a:Tm):Tm =
  a match {
    case Var(n) ⟹ Var(n)
    case Lam(a) ⟹ Lam(norm(a))
    case App(a,b) ⟹ app(norm(a),norm(b))
  }
```

5 Normalization by evaluation

[PE88][BS91]

```
// Normalization by evaluation for untyped lambda calculus


// Lambda terms with named variables

class Tm
case class Var(x:String) extends Tm
case class Lam(x:String, a:Tm) extends Tm
case class App(a:Tm, b:Tm) extends Tm

// HOAS lambda terms

class Sem
case class TmS(a:Tm) extends Sem
case class LamS(f:Sem ⟹ Sem) extends Sem
case class AppS(a:Sem, b:Sem) extends Sem

// Smart constructor for AppS

def appS(a:Sem, b:Sem):Sem =
  a match {
    case LamS(f) ⟹ f(b)
    case _ ⟹ AppS(a,b)
  }

// Normalizing a Sem term is easy

def norm(a:Sem):Sem =
  a match {
    case TmS(a) ⟹ TmS(a)
    case LamS(f) ⟹ LamS(x ⟹ norm(f(x)))
    case AppS(a,b) ⟹ appS(norm(a),norm(b))
```

```
  }

// Conversion from Tm to Sem

def eval(env:Map[String,Sem], a:Tm):Sem =
  a match {
    case Var(x)  ⟹  env(x)
    case Lam(x, a)  ⟹  LamS(v ⟹ eval(env + (x → v), a))
    case App(a,b)  ⟹  AppS(eval(env,a),eval(env,b))
  }

def tmToSem(a:Tm):Sem = eval(Map(),a)

// Conversion from Sem to Tm

var n = 0
def fresh() = { n += 1; s"x$n" }

def reify(a:Sem):Tm =
  a match {
    case TmS(a)  ⟹  a
    case LamS(f)  ⟹  val x = fresh(); Lam(x, reify(f(TmS(Var(x)))))
    case AppS(a,b)  ⟹  App(reify(a),reify(b))
  }

def semToTm(a:Sem):Tm = reify(a)

// Example

val z = LamS(f ⟹ LamS(x ⟹ x))
val s = LamS(n ⟹ LamS(f ⟹ LamS(z ⟹ AppS(AppS(n,f),AppS(f,z)))))

val one = AppS(s,z)
val two = AppS(s,one)

reify(two)
reify(norm(two))
```

6 Conclusion

References

[Brz64] Janusz A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, October 1964. doi:10.1145/321239.321249.

[BS91] U. Berger and H. Schwichtenberg. An inverse of the evaluation functional for typed lambda -calculus. In *[1991] Proceedings Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 203–211, 1991. doi:10.1109/LICS.1991.151645.

[CP95] Cliff Click and Michael Paleczny. A simple graph-based intermediate representation. *SIGPLAN Not.*, 30(3):35–49, March 1995. doi:10.1145/202530.202534.

[KA10] Chantal Keller and Thorsten Altenkirch. Hereditary Substitutions for Simple Types, Formalized. Proceedings of the Mathematically Structured Functional Programming workshop, 2010. URL: https://hal.inria.fr/inria-00520606.

[ORT09] Scott Owens, John Reppy, and Aaron Turon. Regular-expression derivatives re-examined. *Journal of Functional Programming*, 19(2):173–190, 2009. doi:10.1017/S0956796808007090.

[PE88] F. Pfenning and C. Elliott. Higher-order abstract syntax. *SIGPLAN Not.*, 23(7):199–208, June 1988. doi:10.1145/960116.54010.