

BOTTOM-UP REWRITING WITH SMART CONSTRUCTORS, HEREDITARY SUBSTITUTION & NORMALIZATION BY EVALUATION

Jules Jacobs

September 27, 2021

In this note we will investigate these three well-known techniques for rewriting to normal form. We'll look at how to use them to optimize regular expressions and compute β normal forms of lambda terms. We will see that these techniques share the same key idea, but differ in how binders are represented and how substitution is handled:

Hereditary substitution = smart constructors + AST substitution

Normalization by evaluation = smart constructors + HOAS substitution

The formulation of normalization by evaluation in this note is somewhat simpler than the conventional presentation, which (in my view) intertwines conversion to and from HOAS with normalization itself.

CONTENTS

1	Introduction	1
2	Bottom-up rewriting with smart constructors	2
2.1	Smart constructors	3
2.2	Converting to normal form	3
2.3	Handling commutativity	4
2.4	Optimizing at parse time	4
3	Better normal form representations	5
4	Hereditary substitution	6
4.1	De Bruijn indices	8
5	Normalization by evaluation	8
5.1	From ordinary lambda terms to HOAS and back	9
6	Conclusion	10

1 INTRODUCTION

Suppose we want to simplify regular expressions consisting of the following operations. The symbol \emptyset 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:

$$r \in \text{Re} ::= \emptyset \mid 1 \mid 'c' \mid r + r \mid r \cdot r \mid r^*,$$

We want to rewrite by repeatedly using the following equations from left to right:

$r + 0 = r$	$r \cdot 0 = 0$	$0^* = 1$
$0 + r = r$	$0 \cdot r = 0$	$1^* = 1$
$r + r = r$	$r \cdot 1 = r$	$(r^*)^* = r^*$
$(r + s) + t = r + (s + t)$	$1 \cdot r = r$	
	$(r \cdot s) \cdot t = r \cdot (s \cdot t)$	

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

Simplifying regexes using those equations is useful for implementing regular expression matching with Brzozowski derivatives [Brz64, ORTo9]. 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 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 will 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 cdot 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`, and `star` for the ordinary constructors `Seq`, `Alt`, and `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` and `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. Let's call this idea *smart copying*: like a copy function, but calling the smart constructors instead.

```
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 keep repeatedly rewriting using commutativity, 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 $r_1 + r_2 + \dots + r_n$ 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 when converting JVM bytecode to its sea-of-nodes intermediate representation [CP95]. 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 through 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.

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 $(r + s) + t = r + (s + t)$ and commutativity $r + s = s + r$ and idempotence $r + r = r$ into the representation.

```
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
```

We no longer need the `Eps` and `Emp` classes for 0 and 1, because we can represent them as empty alternative and sequence nodes.

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

The smart constructors for `Seq2` and `Alt2` are significantly simpler than those for `Seq` and `Alt`:

```
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 normal form. Alternatively, we could also use the `Re2` representation throughout and remove `Re` entirely, but I show the conversion functions here for illustration:

```

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))
  }

```

Some examples:

```

val a = Chr2('a')
val b = Chr2('b')
val z = alt2(Set(a,b,emp2,eps2)) // Alt2(Set(Chr2(c), Chr2(d), Seq2(List())))
alt2(Set(z,z,a)) // Alt2(Set(Chr2(c), Chr2(d), Seq2(List())))
seq2(List(emp2, a, b)) // Alt2(Set())
re2ToRe(z) // Alt(Eps,Alt(Chr(d),Chr(c)))

```

4 HEREDITARY SUBSTITUTION

While regexes are fun, simplification becomes more interesting when binders are involved, such as with lambda calculus with $\lambda x.e$ and function application $f\ x \equiv \text{app}(f,x)$:

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

We want to simplify with respect to the β -rule that applies a lambda to an argument:

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

We could start with a lambda term and keep applying this rule wherever it applies (and choosing arbitrarily when it applies in multiple places), we'd like to do something like the smart constructors we used for regexes.¹ One method for doing that is called *hereditary substitution* [KA10].

¹ Applying the β -rule repeatedly may not terminate for all lambda terms, such as for the term $\text{app}((\lambda x.\text{app}(x,x)), \lambda x.\text{app}(x,x))$. We will assume that the lambda term that we want to simplify satisfies strong normalization, which means that any order of applying the β -rule will eventually terminate. This is guaranteed if the term type-checks in the simply typed lambda calculus, for instance.

Hereditary substitution can be viewed as a smart constructor for `app`: whenever `app(e1, e2)` sees that `e1 = λx. e` is a lambda term, it will do the substitution instead of constructing an `app` syntax tree node.

To turn this idea into code, we will first need to decide how to represent lambda terms in our program. We could represent variables `x` as strings `"x"`, but this makes it quite difficult to write a correct substitution function in a purely functional way. The difficulty is that we want to apply the `app` rule under lambdas, so the terms we are substituting may have free variables that need to be renamed to avoid name clashes. At first, you may think that giving all variables in the initial lambda term unique names solves the name clash problem, but that isn't the case: because of substitution the terms get copied so the invariant that all names are unique gets violated, and name clashes can still result.

Instead of string names, we're going to use De Bruijn indices [Bru21]. We represent a variable with a number that indicates how many λ nesting levels we need to traverse to find the λ that the variable belongs to. If we have a term $\lambda x. \lambda y. \lambda z. x$ then the De Bruijn index of `x` will be 2, because we need to traverse over the λz and λy in order to find the λx . This way, we do not have to use any variable names, and can simply write $\lambda x. \lambda y. \lambda z. x \equiv \lambda. \lambda. \lambda. 2$. The `app` nodes have no effect on the De Bruijn indices. For instance, the term $\lambda. \lambda. \text{app}(0, \lambda. \text{app}(0, 2))$ means $\lambda x. \lambda y. \text{app}(y, \lambda z. \text{app}(z, x))$. The details of this encoding are not so important and I leave them to the next subsection: the important part is that this allows us to write a substitution function without much trouble.

We first define the syntax tree nodes:

```
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
```

We define a substitution function `subst(e,f)` that substitutes the term `f(i)` for each variable `i` in `e`. This is called a *parallel substitution* because it substitutes terms for all variables simultaneously:

```
def subst(a:Tm, f:Int => Tm):Tm =
  a match {
    case Var(n) => f(n)
    case Lam(a) => Lam(subst(a, liftS(f))) // we will get to the liftS function later
    case App(a,b) => app(subst(a,f), subst(b,f))
  }
```

Notice that the `subst` function is calling the smart constructor `app`, which we still need to define!

In terms of the parallel substitution function `subst(e,f)` we can define `subst0(e,v)` that substitutes only the first variable $0 \mapsto v$ in `e`. Now that variable 0 is gone, we have to decrement all other variable indices:

```
def subst0(e:Tm, v:Tm):Tm = subst(e, (n) => if(n==0) v else Var(n-1))
```

With this substitution function at hand, we can finally define the smart constructor `app`:

```
def app(a:Tm, b:Tm):Tm =
  a match {
    case Lam(e) => subst0(e, b)
    case _ => App(a,b)
  }
```

The mutual recursion between `subst` and `app` ensures that no `app(e1, e2)` term gets created where `e1` is a lambda.

We can "smart copy" a lambda term to β -normalize it:

```
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))
  }
```

That's all there's to it!

4.1 De Bruijn indices

The `liftS` function lifts a substitution over a lambda. To implement this, we first define a renaming function `rename(e, f)` that renames all variables `i` to `f(i)` in `e`:

```
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))
  }
```

Using these, we can implement `shift` and `liftS`:

```
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)))
```

The function `Var` is the identity substitution, so `shift(e, Var)` substitutes `0` to `e` and decrements all other variables by 1, so we can now define `subst0` more concisely as:

```
def subst0(e:Tm, v:Tm):Tm = subst(e, shift(v, Var))
```

5 NORMALIZATION BY EVALUATION

A more funky representation for lambda terms is *higher order abstract syntax* (HOAS) [PE88]. We represent lambda nodes *as its substitution function*: the lambda term $\lambda x.e$ gets represented as a Scala function that does $f(v) = e[x \mapsto v]$. The return value is another lambda term, that is again represented in this HOAS representation. We call the data type for lambda terms in this representation `Sem`:

```
class Sem
case class LamS(f:Sem => Sem) extends Sem
case class AppS(a:Sem, b:Sem) extends Sem
```

Here's an example term $\lambda x.\lambda y.app(y, \lambda z.app(x, z))$:


```
LamS(x => LamS(y => AppS(y, LamS(z => AppS(x,z)))))
```

The smart constructor `app` is now easy to write:

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

The issue is that this doesn't mutually recursively call itself, like the hereditary substitution did. We can fix that by using `appS` instead of `AppS` in the original lambda terms we construct:

```
LamS(x => LamS(y => appS(y, LamS(z => appS(x,z)))))
```

If we do this consistently, then the only place in our program where the constructor `AppS` is called is in `appS`, and there we have made sure that the first argument isn't a lambda, so we're guaranteed to get a β normal form.

Instead of writing our lambda terms using `appS` in the first place, we can write a smart copy function that does it for us:

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

This is called *normalization by evaluation* [BS91].²

5.1 From ordinary lambda terms to HOAS and back

This may all seem incredibly weird, so we're going to write functions that convert from ordinary lambda terms (where variables are represented as strings) to `Sem` terms and back:

```
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
```

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)
```

² Conventionally, normalization by evaluation is intertwined with conversion to and from HOAS. I find that confusing, because it intertwines separate concerns. So I opted to redefine "normalization by evaluation" to mean what the `norm` function does, namely $\text{HOAS} \rightarrow \text{HOAS}$ normalization, and handle conversion to and from HOAS separately. Type directed normalization by evaluation may also be formulated in a style where HOAS is used throughout. One may also still distinguish between HOAS terms and semantic values represented in a representation tailored to normal forms (similar to what we did with regular expression), and have $\text{HOAS} \rightarrow \text{Sem} \rightarrow \text{HOAS}$.

In order to convert from `Sem` to `Tm`, we have to extend the `Sem` data type with one additional constructor, that injects `Tm` terms into the `Sem` data type:³

```
case class TmS(a:Tm) extends Sem
```

The `norm` function is the only function that pattern matches on `Sem`. We extend it by making it do nothing on `TmS` values: `case TmS(a) => TmS(a)`.

We're also going to need a fresh variable name generation facility, because `Sem` values have no variable names whereas `Tm` values do:

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

We can now convert `Sem` values to `Tm` values:

```
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)
```

Here's an 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) /* App(Lam(x1,Lam(x2,Lam(x3,App(App(Var(x1),Var(x2)),App(Var(x2),Var(x3)))))),
               App(Lam(x4,Lam(x5,Lam(x6,App(App(Var(x4),Var(x5)),
               App(Var(x5),Var(x6)))))),
               Lam(x7,Lam(x8,Var(x8))))) */

reify(norm(two)) // Lam(x9,Lam(x10,App(Var(x9),App(Var(x9),Var(x10)))))
```

6 CONCLUSION

We saw that the essence of the techniques discussed is the *smart constructor* that ensures that its return value is in normal form. On top of those, we can build a *smart copy function* that copies a value but constructs the copy using smart constructors, in order to rewrite it to normal form. We saw that hereditary substitution and normalization by evaluation are both smart constructors for `app`, but use a different mechanism to compute the substitution:

Hereditary substitution = smart constructors + AST substitution

Normalization by evaluation = smart constructors + HOAS substitution

³ It is ugly to have to extend the `Sem` type with a new constructor `case class TS(a:T) extends Sem` every time we want to write a function `Sem → T` for a new type `T`. What we *really* want here is a *polymorphic* elimination function `elim(app, lam) : T × T → T` and `lam : (T → T) → T` and gives `elim(app, lam) : Sem → T`. But I'll take the path of least resistance here.

The formulation of normalization by evaluation in this note is somewhat simpler than the conventional presentation, which (in my view) intertwines conversion to and from HOAS with normalization itself.

ACKNOWLEDGEMENTS. I thank Arjen Rouvoet for his suggestions on style and typography.

REFERENCES

- [Bru21] De Bruijn index. *Wikipedia*, August 2021. URL: https://en.wikipedia.org/wiki/De_Bruijn_index.
- [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.