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

Jules Jacobs

September 27, 2021

We will investigate these three well-known techniques for rewriting to normal form, and how to use them to optimize regular expressions and compute β and η 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 key idea is that the smart constructors always create normal forms: we first perform applicable rewrite rules to the new term that we want to construct. By constructing the rewritten term using smart constructors, we get a mutually recursive set of functions that rewrite to normal form. If we have rewrite rules that do substitution, then the substitution function must also construct its new term using the smart constructors, which makes it part of the mutual recursion.

The formulation of normalization by evaluation in this note is slightly simpler than some conventional presentations, which (in my view) intertwine conversion to and from HOAS with normalization itself.[1]

## CONTENTS



1 E.g., the one on wikipedia `https://en.wikipedia.org/wiki/normalization_by_evaluation`. Note how `reify` does both HOAS → AST conversion and η-long normal form creation, and `meaning` does both AST → HOAS conversion and evaluation. Some approaches, such as *finally tagless* [CKS07] do use HOAS throughout.

## 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 by repeatedly using the following equations from left to right:

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

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

Simplifying regexes using those equations is useful for implementing regular expression matching with Brzozowski derivatives [Brz64, ORT09]. 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:

```
enum Re:
  case Emp // 0
  case Eps // 1
  case Chr(c:Char) // 'c'
  case Seq(a:Re, b:Re) // r · s
  case Alt(a:Re, b:Re) // r + s
  case Star(a: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 (Re.Emp,_) => Re.Emp
    case (_,Re.Emp) => Re.Emp
    case (Re.Eps,x) => x
    case (x,Re.Eps) => x
    case (Re.Seq(x,y),b) => seq(x,seq(y,b))
    case _ => Re.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 (Re.Emp,x) => x
    case (x,Re.Emp) => x
    case (Re.Alt(x,y),b) => alt(x,alt(y,b))
    case _ => if(a==b) a else Re.Alt(a,b)
  }

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

val r = Re.Alt(Re.Star(Re.Star(Re.Seq(Re.Chr('a'),Re.Star(Re.Emp)))),Re.Emp)
norm(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 $r_1 + r_2 + \cdots + r_n$ 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 (Re.Emp,x) => x
  case (x,Re.Emp) => x
  case (Re.Alt(x,y),b) => alt1(x,alt1(y,b))
  case (a,Re.Alt(x,y)) =>
   if(a==x) b
   else if(a.hashCode() < x.hashCode()) Re.Alt(a,b)
   else alt1(x,alt1(a,y))
  case _ =>
   if(a==b) a
   else if(a.hashCode() < b.hashCode()) Re.Alt(a,b)
   else Re.Alt(b,a)
 }
```

This smart constructor is able to do that optimization:

```
val a = Re.Chr('a')
```

```
val b = Re.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 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.

## 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.

```
enum Re2:
  case Chr(a:Char)
  case Seq(rs:List[Re2])
  case Alt(rs:Set[Re2])
  case Star(r: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 = Re2.Alt(Set())
val eps2 = Re2.Seq(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 Re2.Seq(rs) => rs; case x => List(x)}
  if(rs2.contains(emp2)) emp2
  else if(rs2.size == 1) rs2.head
  else Re2.Seq(rs2)
}

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

```
}

def star2(a:Re2):Re2 =
  a match {
    case Re2.Alt(rs) if rs.isEmpty => eps2
    case Re2.Seq(rs) if rs.isEmpty => eps2
    case Re2.Star(_) => a
    case _ => Re2.Star(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. We use the name `norm` : `Re` → `Re2` because normalization is now done by conversion to the tailored normal form representation:

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

We use the name `reify` : `Re2` → `Re` because this conversion reifies the normal form representation back to the original syntax:

```
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 reify(r:Re2):Re =
  r match {
    case Re2.Chr(c) => Re.Chr(c)
    case Re2.Seq(rs) => fold1(rs.map(reify), Re.Eps, Re.Seq)
    case Re2.Alt(rs) => fold1(rs.map(reify), Re.Eps, Re.Alt)
    case Re2.Star(r) => Re.Star(reify(r))
  }
```

Some examples:

```
val c = Re2.Chr('c')
val d = Re2.Chr('d')
val z = alt2(Set(c,d,emp2,eps2))
alt2(Set(z,z,c)) // Alt(Set(Chr(c), Chr(d), Seq(List())))
seq2(List(emp2, c, d)) // Alt(Set())
reify(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 $f\ x \equiv \mathsf{app}(f, x)$:

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

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

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

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.[2] One method for doing that is called *hereditary substitution* [KA10].

Hereditary substitution can be viewed as a smart constructor for app: whenever $\mathsf{app}(e_1, e_2)$ sees that $e_1 = \lambda x, e$ 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 $\lambda$ nesting levels we need to traverse to find the $\lambda$ 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 $\lambda z$ and $\lambda y$ in order to find the $\lambda x$. 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 $\lambda.\lambda.\mathsf{app}(0, \lambda.\mathsf{app}(0, 2))$ means $\lambda x.\lambda y.\mathsf{app}(y, \lambda z.\mathsf{app}(z, x))$. 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 De Bruijn syntax tree nodes:

```
enum Db:
  case Var(x:Int)
  case Lam(a:Db)
  case App(a:Db, b:Db)
```

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:Db, f:Int => Db):Db =
  a match {
```

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

```
    case Db.Var(n) => f(n)
    case Db.Lam(a) => Db.Lam(subst(a,liftS(f))) // we will get to the liftS function later
    case Db.App(a,b) => Db.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:Db, v:Db):Db = subst(e, (n) => if(n==0) v else Db.Var(n-1))
```

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

```
def app(a:Db, b:Db):Db =
  a match {
    case Db.Lam(e) => subst1(e, b)
    case _ => Db.App(a,b)
  }
```

The mutual recursion between `subst` and `app` ensures that no $\mathtt{app}(e_1, e_2)$ term gets created where $e_1$ is a lambda.

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

```
def norm(a:Db):Db =
  a match {
    case Db.Var(n) => Db.Var(n)
    case Db.Lam(a) => Db.Lam(norm(a))
    case Db.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:Db, f:Int => Int):Db =
  a match {
    case Db.Var(n) => Db.Var(f(n))
    case Db.Lam(a) => Db.Lam(rename(a,liftR(f)))
    case Db.App(a,b) => Db.App(rename(a,f),rename(b,f))
  }
```

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

```
def shift(e:Db, f:Int => Db):Int => Db =
  (n) => if(n==0) e else f(n-1)

def liftS(f : Int => Db):Int => Db =
  shift(Db.Var(0), k => rename(f(k), (_+1)))
```

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

```
def subst0(e:Db, v:Db):Db = subst(e, shift(v,Db.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 `Hs`:

```
enum Hs:
  case Lam(f:Hs => Hs)
  case App(a:Hs, b:Hs)
```

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

```
Hs.Lam(x => Hs.Lam(y => Hs.App(y, Hs.Lam(z => Hs.App(x,z)))))
```

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

```
def app(a:Hs, b:Hs):Hs =
  a match {
    case Hs.Lam(f) => f(b)
    case _ => Hs.App(a,b)
  }
```

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

```
Hs.Lam(x => Hs.Lam(y => app(y, Hs.Lam(z => app(x,z)))))
```

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

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

```
def norm(a:Hs):Hs =
  a match {
    case Hs.Lam(f) => Hs.Lam(x => norm(f(x)))
    case Hs.App(a,b) => app(norm(a),norm(b))
  }
```

This is called *normalization by evaluation* [BS91].[3]

3 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 HOAS → 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 in a representation tailored to normal forms (similar to what we did with regular expression), and have HOAS → Sem → HOAS.

## 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 `Hs` terms and back:

```
enum Tm:
  case Var(x:String)
  case Lam(x:String, a:Tm)
  case App(a:Tm, b:Tm)
```

Conversion from `Tm` to `Hs`:

```
def eval(env:Map[String,Hs], a:Tm):Hs =
  a match {
    case Tm.Var(x) => env(x)
    case Tm.Lam(x, a) => Hs.Lam(v => eval(env + (x → v), a))
    case Tm.App(a,b) => Hs.App(eval(env,a),eval(env,b))
  }

def toHs(a:Tm):Hs = eval(Map(),a)
```

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

```
case class ResTm(a:Tm) extends Hs
```

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

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

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

```
def freshLam(f:Tm => Tm):Tm = { val x = fresh(); Tm.Lam(x, f(Tm.Var(x))) }

def toTm(a:Hs):Tm =
  a match {
    case Hs.ResTm(a) => a
    case Hs.Lam(f) => freshLam(x => toTm(f(Hs.ResTm(x))))
    case Hs.App(a,b) => Tm.App(toTm(a),toTm(b))
  }
```

Here's an example:

```
val zero = Hs.Lam(f => Hs.Lam(x => x))
val succ = Hs.Lam(n => Hs.Lam(f => Hs.Lam(z => Hs.App(Hs.App(n,f),Hs.App(f,z)))))

val one = Hs.App(succ,zero)
val two = Hs.App(succ,one)

toTm(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))))) */

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

## 5.2 *Folding HOAS & finally tagless*

The `toTm : Hs ⟶ Tm` and `norm : Hs ⟶ Hs` functions are somewhat similar. We can factor out the pattern by defining a `fold` function on `Hs`, with which we can write functions `Hs ⟶ T` for any `T`. The `fold` function takes smart constructors `app : T × T ⟶ T` and `lam : (T ⟶ T) ⟶ T` as arguments, and copies the `Hs` term with them to obtain a value of type `T`. In order to write `fold`, we need to extend `Hs` with a new constructor, as we did before:

```
case class Res(a:Object) extends Hs // hack for fold
```

We can then write `fold`:

```
def fold[T](a:Hs, app : (T,T) => T, lam : (T => T) => T) : T =
  a match {
    case Hs.Res(x) => x.asInstanceOf[T]
    case Hs.Lam(f) => lam(t => fold(f(Hs.Res(t.asInstanceOf[Object])), app, lam))
    case Hs.App(a,b) => app(fold(a, app, lam), fold(b, app, lam))
  }
```

As you can see, this a bit of a hacky approach that requires unsafe casts. It has the advantage of being fairly simple and require minimal changes to what we already have. See [WW03] for a non-hacky approach.

We can use `fold` to write `toTm` and `norm` as one-liners:

```
def toTm(a:Hs):Tm = fold[Tm](a, Tm.App, freshLam)
def norm(a:Hs):Hs = fold[Hs](a, app, Hs.Lam)
```

An alternative is to represent `a:Hs` values *as their folding function* `(app,lam) => fold(a, app, lam)`. This approach is called *finally tagless* [CKS07] and it works very nicely. I would highly recommend checking out that paper.

# 6 TYPE DIRECTED NORMALIZATION BY EVALUATION

Type directed normalization by evaluation can be used to put values in η expanded form. In η expanded form we are only allowed to use variables $f : A \longrightarrow B$ of function type as the first argument of app. To use $f$ in any other place then we must instead write $f \equiv \lambda x.\mathsf{app}(f, x)$. We need type information because the types determine how much we η expand:

- Expanding $\lambda x.x$ at type $A \longrightarrow A$ gives $\lambda x.x$
- Expanding $\lambda x.x$ at type $(A \longrightarrow A) \longrightarrow (A \longrightarrow A)$ gives $\lambda f.\lambda x.\mathsf{app}(f, x)$

We start by defining a data type for types:

```
enum Ty:
  case Base
  case Arrow(a:Ty, b:Ty)
```

Instead of `norm : Hs ⟶ Hs`, we're going to use a tailored representation for normal forms, like we did for regular expressions. We're going to have `norm : Hs ⟶ Sem` and `reify : Sem ⟶ Hs`, where `Sem` is the "semantic domain" tailored to representing normal forms.

In fact, the situation is a little bit more interesting. The semantic domain is going to depend on the type of values we're representing: the domain $\mathsf{Sem}_A$ is going to be indexed by the type $A$ in the following way:

$$\mathsf{Sem}_{\mathsf{Base}} := \mathsf{Hs}$$
$$\mathsf{Sem}_{A \to B} := \mathsf{Sem}_A \longrightarrow \mathsf{Sem}_B$$

In other words, the semantic domain of base types is HOAS terms `Hs`, but the semantic domain for function types is going to be Scala functions between the respective semantic domains. Unfortunately, we are not using Agda, and my Scala knowledge is insufficient to know if it is possible to do this in Scala. In any case I'd like to keep the code here simple, and make it possible to translate into other languages without Agda's (or Scala's) fancy type system. So we're going to smush the semantic domain together into a single type, and we're going to put the dependent types in comments:

```
enum Sem:
  case Syn(a:Hs) // Sem_Base = syntactic terms Hs
  case Lam(f:Sem => Sem) // Sem_Arrow(A,B) = Sem_A ⟶ Sem_B
```

Although the `Sem` domain doesn't have a case for `App`, we *can* write a smart constructor for it:

```
  // Smart constructor
  // appS : Sem_{A ⟶ B} ⟶ Sem_A ⟶ Sem_B
  def appS(a:Sem, b:Sem):Sem =
    a match {
      case Sem.Lam(f) => f(b)
      // Types guarantee that we don't need any other case!
    }
```

Because we assume that `a` is of function type, we can ignore the `Syn` case. We'll need to be careful to only pass `a`'s of function type into `appS`, of course.

We can write `norm` : `Hs` ⟶ `Sem` as a fold using this smart constructor:

```
// norm : Hs_t ⟶ Sem_t
def norm(x:Hs):Sem = fold[Sem](x, appS, Sem.Lam)
```

The more complicated direction turns out to be reifying `Sem` values back to syntactic `Hs` values:

```
// reify : Sem_t ⟶ Hs_t
def reify(t:Ty, x:Sem):Hs =
  (t,x) match {
    case (Ty.Arrow(a,b), Sem.Lam(f)) => Hs.Lam(y => reify(b, f(reflect(a, y))))
    case (Ty.Base, Sem.Syn(y)) => y
  }

// does eta expansion
// reflect : Hs_t ⟶ Sem_t
def reflect(t:Ty, x:Hs):Sem =
  t match {
    case Ty.Arrow(a,b) => Sem.Lam(y => reflect(b, Hs.App(x, reify(a,y))))
    case Ty.Base => Sem.Syn(x)
  }
```

Because the `Sem` domain only contains function values for function types, we are *forced* to do η-expansion: there is simply no way to represent a plain variable $f$ in the semantic domain at function type, we must write $\lambda x.\mathsf{app}(f, x)$.

By normalising and then reifying, we can do typed normalization by evaluation from HOAS terms to HOAS terms:

```
// nbe : Hs_t ⟶ Hs_t
def nbe(t:Ty, e:Hs) = reify(t, norm(e))
```

By using the conversion functions we can also get normalization by evaluation on ordinary terms:

```
// nbeT : Tm_t ⟶ Tm_t
def nbeT(t:Ty, e:Tm) = toTm(nbe(t,toHs(e)))
```

Here's an example that normalizes SK combinators [NbE21]:

```
val k = Hs.Lam(x => Hs.Lam(y => x))
val s = Hs.Lam(x => Hs.Lam(y => Hs.Lam(z => Hs.App(Hs.App(x,z), Hs.App(y,z)))))
val skk = Hs.App(Hs.App(s,k),k)

toTm(skk) /* App(App(Lam(x11,Lam(x12,Lam(x13,
              App(App(Var(x11),Var(x13)),App(Var(x12),Var(x13)))))),
                Lam(x14,Lam(x15,Var(x14)))), Lam(x16,Lam(x17,Var(x16)))) */

val tb = Ty.Arrow(Ty.Base,Ty.Base)

// normalizes to the identify function
toTm(nbe(tb, skk)) // Lam(x18,Var(x18))

val tbb = Ty.Arrow(tb, tb)

// at higher type, it eta expands the identity function
toTm(nbe(tbb, skk)) // Lam(x19,Lam(x20,App(Var(x19),Var(x20))))
```

I would highly recommend looking at Sam Lindley's slide deck [Lin16] if you want to learn more about typed normalization by evaluation. It is excellent and explains simple normalization by evaluation very clearly, but also goes much beyond this.

## 7 SUMMARY

We saw that the essence of these techniques 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* `norm` that copies a value but constructs the copy using smart constructors, in order to rewrite it to normal form. The smart copy function is the same as `fold`*ing* the data type with the smart constructors. The smart constructors do not have to use the original syntactic representation: we can use a *tailored normal form representation* as we did with regular expressions with `Seq` represented as a list and `Alt` as as set. In this case, the smart copy function has the syntactic representation type as input and the normal form representation type as output. We can `reify` the normal form representation back into the syntactic representation.

Hereditary substitution and untyped normalization by evaluation are both based on smart constructors for app, but use a different representation and mechanism to compute the substitution. Typed normalization by evaluation uses a separate semantic domain to tailored to representing normal forms. This semantic domain is indexed by the type of the expressions. We can still separate the AST ↔ HOAS conversion and HOAS ↔ Normal Form transformation, which makes each individual piece a bit simpler.

ACKNOWLEDGEMENTS. I thank Arjen Rouvoet for his suggestions on style and typography, and Molossus Spookee, Paolo Giarrusso, and Edwin Brady for their suggestions.

## 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.

[CKS07] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. Finally tagless, partially evaluated. In Zhong Shao, editor, *Programming Languages and Systems*, pages 222–238, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[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.

[Lin16] Sam Lindley. Normalization by evaluation. 2016. URL: http://homepages.inf.ed.ac.uk/slindley/nbe/nbe-cambridge2016.pdf.

[NbE21] Normalization by evaluation. *Wikipedia*, August 2021. URL: https://en.wikipedia.org/wiki/Normalisation_by_evaluation.

[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.

[WW03] Geoffrey Washburn and Stephanie Weirich. Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism. 38(9):249–262, August 2003. doi:10.1145/944746.944728.