A tale of four lambda-calculus interpreters

Stephanie Weirich
with Noé De Santo
May 2025

Penn Engineering
University of Pennsylvania

Scope-Safe Lambda-Calculus in Haskell

```
data Nat = Z | S Nat

data Fin n where              -- Bounded natural number
  FZ :: Fin (S n)
  FS :: Fin n -> Fin (S n)

data Bind n where
  Bind :: Exp (S n) -> Bind n   -- Increases the scope by one

data Exp n where
  Var :: (Fin n) -> Exp n
  Lam :: (Bind n) -> Exp n
  App :: (Exp n) -> (Exp n) -> Exp n
```

1. Environment-based Interpreter

```
-- finite map from indices to values
type Env n = Fin n -> Val
-- empty environment
nil :: Env Z
-- extend environment
(.:) :: Val -> Env n -> Env (S n)
data Val where
  VLam :: Env n -> Bind n -> Val

eval :: Env m -> Exp m -> Val
eval r (Var x) = r x
eval r (Lam e) = VLam r e
eval r (App f a) =
  case eval r f of
    VLam r' b -> eval (eval r a .: r') b
```

- Scoping ensures that all variable references are in the domain of the environment
- VLam is a closure: evaluating a lambda expression saves the current environment
- In an application, add the value of the argument to the saved environment

Is this evaluator call-by-value, call-by-name, or something else?

2. Substitution-based Interpreter

```
eval :: Exp Z -> Exp Z
eval (Var x) = case x of {}
eval (Lam b) = Lam b
eval (App f a) =
  case eval f of
    Lam b ->
      eval (instantiate b (eval a))

instantiate :: Bind n -> Exp n -> Exp n
instantiate (Bind b) a =
   subst (singleton a) b
```

- Scoping ensures that the variable case is unreachable
- In application, substitute the evaluation of the argument for the bound variable in the binder
- Need to define substitution

Is this evaluator call-by-value, call-by-name, or something else?

Substitutions

```
type Subst m n = Fin m -> Exp n

-- empty substitution
nil :: Subst Z n
-- extend substitution
(.:) :: Exp n -> Subst m n
        -> Subst (S m) n

-- identity
id :: Subst n n
id = Var
-- composition
(⊙) :: Subst m n -> Subst n p
        -> Subst m p
```

A (parallel) substitution applies to all indices in the current scope (m) and produces expressions in a new scope (n).

It can be constructed like a list.
It can be composed like a function.

```
-- decrease scope
singleton :: Exp n -> Subst (S n) n
singleton a = a .: id

-- increase scope
shift :: Subst n (S n)
shift = \x -> Var (1 + x)
```

Substitution in expressions

```
data Bind n where
  Bind :: Exp (S n) -> Bind n

subst :: Subst m n -> Exp m -> Exp n
subst r (Var x) = r x
subst r (Lam (Bind b)) =
  Lam (Bind (subst (up r) b))
subst r (App a1 a2) =
  App (subst r a1) (subst r a2)

-- go under a binder
up :: Env m n → Subst (S m) (S n)
up r = Var 0 .: r ⊙ shift
```

Apply substitution in variable case
"Lift" substitution at binders

Leave variable 0 alone, shift all other variables to new scope

Lennart's Benchmark

```
let Zero = \z.\s.z;
   Succ = \n.\z.\s.s n;
   one = Succ Zero;
   two = Succ one;
   three = Succ two;
   isZero = \n.n true (\m.false);
   const = \x.\y.x;
   Pair = \a.\b.\p.p a b;
   fst = \ab.ab (\a.\b.a);
   snd = \ab.ab (\a.\b.b);
   fix = \g. (\ x. g (x x)) (\ x. g (x x));
   add = fix (\radd.\x.\y. x y (\ n. Succ (radd n y)));
   mul = fix (\rmul.\x.\y. x Zero (\ n. add y (rmul n y)));
   fac = fix (\rfac.\x. x one (\ n. mul x (rfac n)));
   eqnat = fix (\reqnat.\x.\y. x (y true (const false))
            (\x1.y false (\y1.reqnat x1 y1)));
   sumto = fix (\rsumto.\x. x Zero (\n.add x (rsumto n)));
   n5 = add two three;
   n6 = add three three;
   n17 = add n6 (add n6 n5);
   n37 = Succ (mul n6 n6);
   n703 = sumto n37;
   n720 = fac n6
in eqnat n720 (add n703 n17)
```

- Adapted from Lennart's "Lambda-calculus cooked four ways"
- Scott encoding of
  6! == sum [0.. 37] + 17
- Needs 119694 beta-reductions
- Benchmarking suite: github.com/sweirich/lambda-n-ways

Substitution-based: 3.01 s
Environment-based: .000312 s

Comparison

Environment-based

- More efficient
- Implementation is shorter
- Safer (no error case)

```
type Env n = Fin n -> Val
```

Substitution-based

- Looks like a POPL paper
- Don't need a new type for values
- **Extends to open terms**

```
type Subst m n = Fin m -> Exp n
```

Key data structures are remarkably similar!

Can we transfer key ideas from the environment-based interpreter to the substitution-based interpreter?

1. Environment-based Interpreter: Two key ideas

```
type Env n = Fin n -> Val
nil  :: Env Z
(.:) :: Val -> Env n -> Env (S n)
data Val where
  VLam :: Env n -> Bind n -> Val

eval :: Env n -> Exp n -> Val
eval r (Var x) = r x
eval r (Lam b) = VLam r b
eval r (App a1 a2) =
  case eval r a1 of
    VLam s (Bind b) ->
      eval (eval r a2 .: s) b
```

1. Suspend the environment when evaluating lambda expressions and extend saved environment in applications.

2. Pass the environment explicitly to avoid re-evaluating arguments

Eager vs. **Delayed** Substitution at Binding

```
data Bind n where
  Bind :: Exp (S n) -> Bind n

subst :: Subst m n -> Exp m -> Exp n
subst r (Var x) = r x
subst r (Lam (Bind b)) =
  Lam (Bind (subst (up r) b))
subst r (App a1 a2) =
  App (subst r a1) (subst r a2)

-- go under a binder
up :: Env m n → Subst (S m) (S n)
up e = var 0 . : e ⨀ shift
```

```
data Bind n where
  Bind :: Subst m n -> Exp (S m) -> Bind n

subst :: Subst m n -> Exp m -> Exp n
subst r (Var x) = r x
subst r (Lam (Bind r' b)) =
  Lam (Bind (r’ ⨀ r) b)
subst r (App a1 a2) =
  App (subst r a1) (subst r a2)
```

Eager vs. **Delayed** Substitution Evaluator

```
eval :: Exp Z -> Exp Z
eval (Var x) = case x of {}
eval (Lam b) = Lam b
eval (App a1 a2) =
  case eval a1 of
    Lam b ->
      eval (instantiate b (eval a2))
    _ -> error "should be a lambda"

instantiate :: Bind n -> Exp n -> Exp n
instantiate (Bind b) a =
   subst (a .: id) b
```

```
eval :: Exp Z -> Exp Z
eval (Var x) = case x of {}
eval (Lam b) = Lam b
eval (App a1 a2) =
  case eval a1 of
    Lam b ->
      eval (instantiate b (eval a2))
    _ -> error "should be a lambda"

instantiate :: Bind n -> Exp n -> Exp n
instantiate (Bind r b) a =
      subst (a .: r) b
```

3. **Delayed** Substitution Evaluator

```
eval :: Exp Z -> Exp Z
eval (Var x) = case x of {}
eval (Lam b) = Lam b
eval (App a1 a2) =
  case eval a1 of
    Lam (Bind r b) ->
      eval (subst (eval a2 .: r) b)
    _ -> error "should be a lambda"
```

Inline definition of instantiate

Can we delay this substitution too?

4. **Explicit** Delayed Substitution Evaluator

```
eval :: Subst m Z -> Exp m -> Exp Z
eval r (Var x) = r x
eval r (Lam b) = subst r (Lam b)
eval r (App a1 a2) =
  case eval r a1 of
    Lam (Bind s b) ->
      eval (eval r a2 .: s) b
    _ -> error "should be a lambda"
```

Continue delayed substitution through the entire term

Substitute into the body of the binder while evaluating it

Lennart’s Benchmark

```
let Zero = \z.\s.z;
   Succ = \n.\z.\s.s n;
   one = Succ Zero;
   two = Succ one;
   three = Succ two;
   isZero = \n.n true (\m.false);
   const = \x.\y.x;
   Pair = \a.\b.\p.p a b;
   fst = \ab.ab (\a.\b.a);
   snd = \ab.ab (\a.\b.b);
   fix = \g. (\ x. g (x x)) (\ x. g (x x));
   add = fix (\radd.\x.\y. x y (\ n. Succ (radd n y)));
   mul = fix (\rmul.\x.\y. x Zero (\ n. add y (rmul n y)));
   fac = fix (\rfac.\x. x one (\ n. mul x (rfac n)));
   eqnat = fix (\reqnat.\x.\y. x (y true (const false))
             (\x1.y false (\y1.reqnat x1 y1)));
   sumto = fix (\rsumto.\x. x Zero (\n.add x (rsumto n)));
   n5 = add two three;
   n6 = add three three;
   n17 = add n6 (add n6 n5);
   n37 = Succ (mul n6 n6);
   n703 = sumto n37;
   n720 = fac n6
in eqnat n720 (add n703 n17)
```

- Adapted from Lennart’s “Lambda-calculus cooked four ways”
- Scott encoding of
  6! == sum [0.. 37] + 17
- Needs 119694 beta-reductions
- Benchmarking suite: github.com/sweirich/lambda-n-ways

Substitution-based: 3.01 s
Environment-based: 312 mu s
Delayed binder: 801 mu s
Environment-passing: 566 mu s

Penn Engineering
University of Pennsylvania

Can we bottle this up? Yes: "autoenv" library

```
type Env v m n       -- Abstract type
(!) :: (SubstVar v) ⇒ Env v m n → Fin m → v n

class SubstVar v ⇒ Subst v e where
  -- substitute with v in e
     applyE :: Env v m n → e m → e n
class Subst v v ⇒ SubstVar v where
  -- variable constructor / identity substitution
  var :: Fin n → v n

data Bind v e n      -- Abstract type
instance SubstVar v ⇒ Subst v (Bind v e)
instantiate :: Bind v e n → Exp n → Exp n
```

3. Client code – Version 3

```
data Exp :: Nat -> Type where
  Var :: Fin n -> Exp n
  Lam :: Bind Exp Exp n -> Exp n
  App :: Exp n -> Exp n -> Exp n

instance SubstVar Exp where var = Var
instance Subst Exp Exp where
  subst r (Var x) = r ! x
  subst r (Lam b) = Lam (subst r b)
  subst r (App a1 a2) = App (subst r a1) (subst r a2)

eval :: Exp Z -> Exp Z            -- Version 3
eval (Var x) = case x of {}
eval (Lam b) = Lam b
eval (App a1 a2) =
  case eval a1 of
     Lam b ->
        eval (instantiate b (eval a2))
     _ -> error "should be a lambda”
```

4. Client code – Version 4

```
data Exp :: Nat -> Type where
  Var :: Fin n -> Exp n
  Lam :: Bind Exp Exp n -> Exp n
  App :: Exp n -> Exp n -> Exp n

instance SubstVar Exp where
  var = Var
instance Subst Exp Exp where – uses GHC.Generics

eval :: Env m Z -> Exp m -> Exp Z            -- Version 4
eval r (Var x) = case x of {}
eval r (Lam b) = Lam (applyE r b)
eval r (App a1 a2) =
  case eval r a1 of
     Lam b ->
        instantiateWith eval b (eval r a2)
     _ -> error "should be a lambda”
```

Suggested Questions

1. Normalization for open terms?
2. How expressive is the library?
3. Do the dependent types help?
4. Can you optimize the representation of environments?

https://github.com/sweirich/autoenv
https://github.com/sweirich/lambda-n-ways

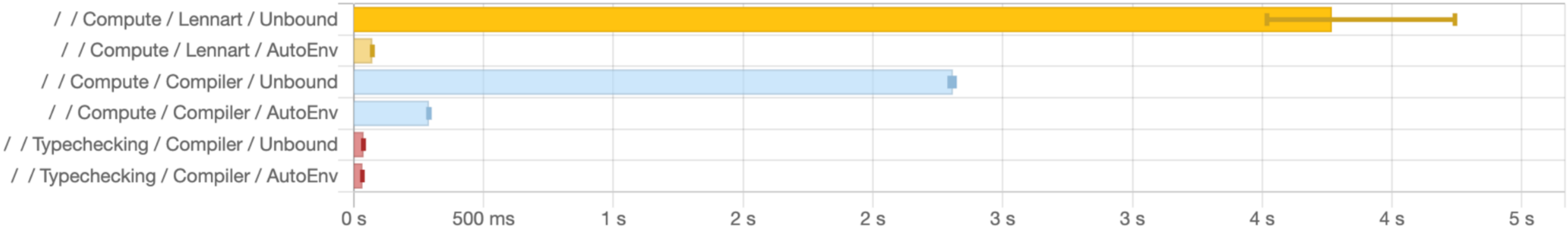
How expressive is this library?

- Examples for various forms of binding
  - binders with names for printing
  - n-ary binders
  - pattern binders
  - recursive binders
  - dependent, telescoped binders
- Working in a ”ScopedMonad” helps
  - keeps track of current scope and associated data (i.e. names for printing, types, etc.)
- Ported pi-forall type checker to use this library (benchmarks in progress)

https://github.com/Ef55/pi-forall

Initial pi-forall performance comparison

- Compute/Lennart : The usual Lennart benchmark, but in pi-forall
- Compute/Compiler : Check that the result of interpreting an expr yields the same result as compiling and then interpreting the stack program
- Typechecking/Compiler: Typechecking of a tiny intrinsically typed compiler (from exprs to stack machine)



Dependent types yea or nah?

- Am I really going to argue for weaker types?
  I would not use de Bruijn indices without static scoping.
  Caveat: multiple scopes are challenging (e.g. System F)
- Need two properties about natural numbers.
  Have been using explicitly, could use solver plugin.
- Even with dependent types, I still made mistakes.
  pi-forall needs both these operations, easily confused.

```
weaken :: Fin n -> Fin (S n)   -- identity function
shift  :: Fin n -> Fin (S n)   -- increment
```

Defunctionalized, strict(er) environment

```
data Env (a :: Nat -> Type) (n :: Nat) (m :: Nat) where
    Weak  :: (SNat m) -> Env a n (m + n)
    Inc   :: (SNat m) -> Env a n (m + n) -- shift by m
    Cons  :: (a m) -> (Env a n m) -> Env a ('S n) m -- extend
    (:<>) :: (Env a m n) -> (Env a n p) -> Env a m p – compose
```

```
data Env (a :: Nat -> Type) (n :: Nat) (m :: Nat) where
  Weak  :: !(SNat m) -> Env a n (m + n)
  Inc   :: !(SNat m) -> Env a n (m + n) -- shift by m
  Cons  :: (a m) -> !(Env a n m) -> Env a ('S n) m -- extend
  (:<>) :: !(Env a m n) -> !(Env a n p) -> Env a m p -- compose
```

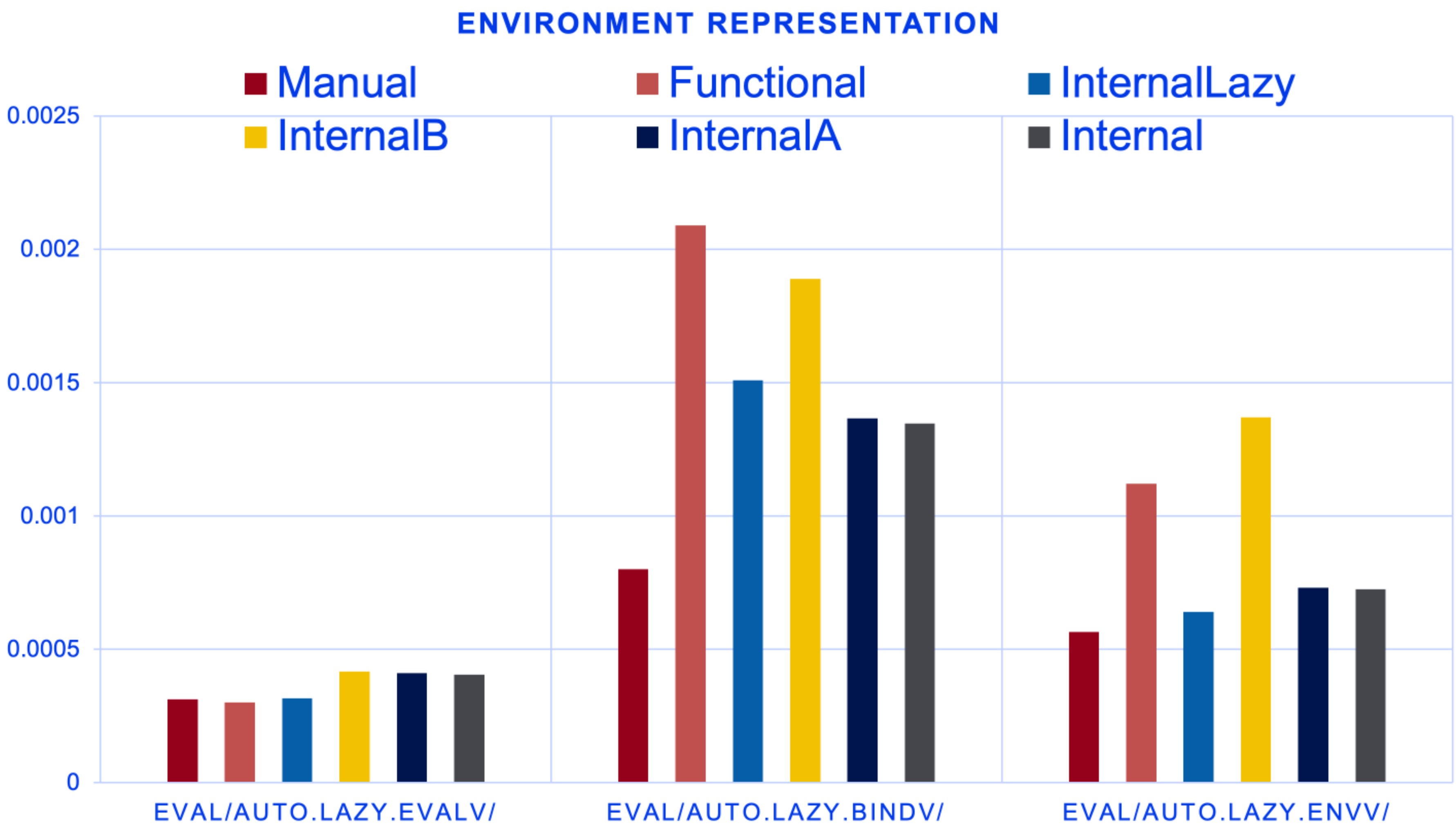
Smart Composition

```
comp :: forall a m n p. SubstVar a =>
    Env a m n -> Env a n p -> Env a m p
comp (Weak (k1 :: SNat m1)) (Weak (k2 :: SNat m2)) =
  case axiomAssoc @m2 @m1 @m of Refl -> Weak (sPlus k2 k1)
comp (Weak 0) s = s   -- Weaken by 0 is identity
comp s (Weak 0) = s
comp (Inc (k1 :: SNat m1)) (Inc (k2 :: SNat m2)) =
  case axiomAssoc @m2 @m1 @m of Refl -> Inc (sPlus k2 k1)
comp s (Inc 0) = s
comp (Inc 0) s = s
comp (Inc (1 + n)) (Cons _ p) = comp (Inc n) p
comp (s1 :<> s2) s3 = comp s1 (comp s2 s3) -- reassociate
comp (Cons t s1) s2 = Cons (applyE s2 t) (comp s1 s2)
comp s1 s2 = s1 :<> s2
```

How to represent environments?

- Functional: Fin n -> v m
- InternalB: Defunctionalized, strict
  - Optimize: applyE idE x = x
- InternalA: Defunctionalized, strict
  - Optimize: smart composition
- InternalLazy: Defunctionalized, lazy
  - both A and B
- Internal: Defunctionalized, strict
  - both A and B

(Note: Vectors or other “sequence” data structures requires a slightly different interface for the library.)


ENVIRONMENT REPRESENTATION
Manual
Functional
InternalLazy
InternalB
InternalA
Internal
0.0025
0.002
0.0015
0.001
0.0005
0
EVAL/AUTO.LAZY.EVALV/
EVAL/AUTO.LAZY.BINDV/
EVAL/AUTO.LAZY.ENVV/

Conclusion

https://github.com/sweirich/autoenv

https://github.com/sweirich/lambda-n-ways

https://github.com/Ef55/pi-forall

Penn Engineering
University of Pennsylvania

Penn Engineering
UNIVERSITY of PENNSYLVANIA