How to Implement the Lambda Calculus, Quickly

- Stephanie Weirich
- University of Pennsylvania

- Haskell Love
- September 10, 2021

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

Key lambda-calculus operation

Capture-avoiding substitution

$$a \{ b / x \}$$

$$(z\ (\lambda y.z))\ \{ y / z \}$$

$$\Longmapsto$$

$$y\ (\lambda w.\ y)$$

Klaus Ostermann
@klauso3

...

What I tell others what I do: Design cool programming languages. What I really do: Debug substitution functions.

6:08 AM · Sep 3, 2021 · Twitter Web App

8 Retweets **1** Quote Tweet **76** Likes

Magic implementation?

```
data Exp = Var Var                 -- x
         | Lam (Bind Exp)          -- λx.a
         | App Exp Exp             -- (a b)
           deriving (Substitution)

-- definition of substitution "for free"
-- subst :: Var -> Exp -> Exp -> Exp  -- a { b / x }
```

Binding Library (unbound-generics)

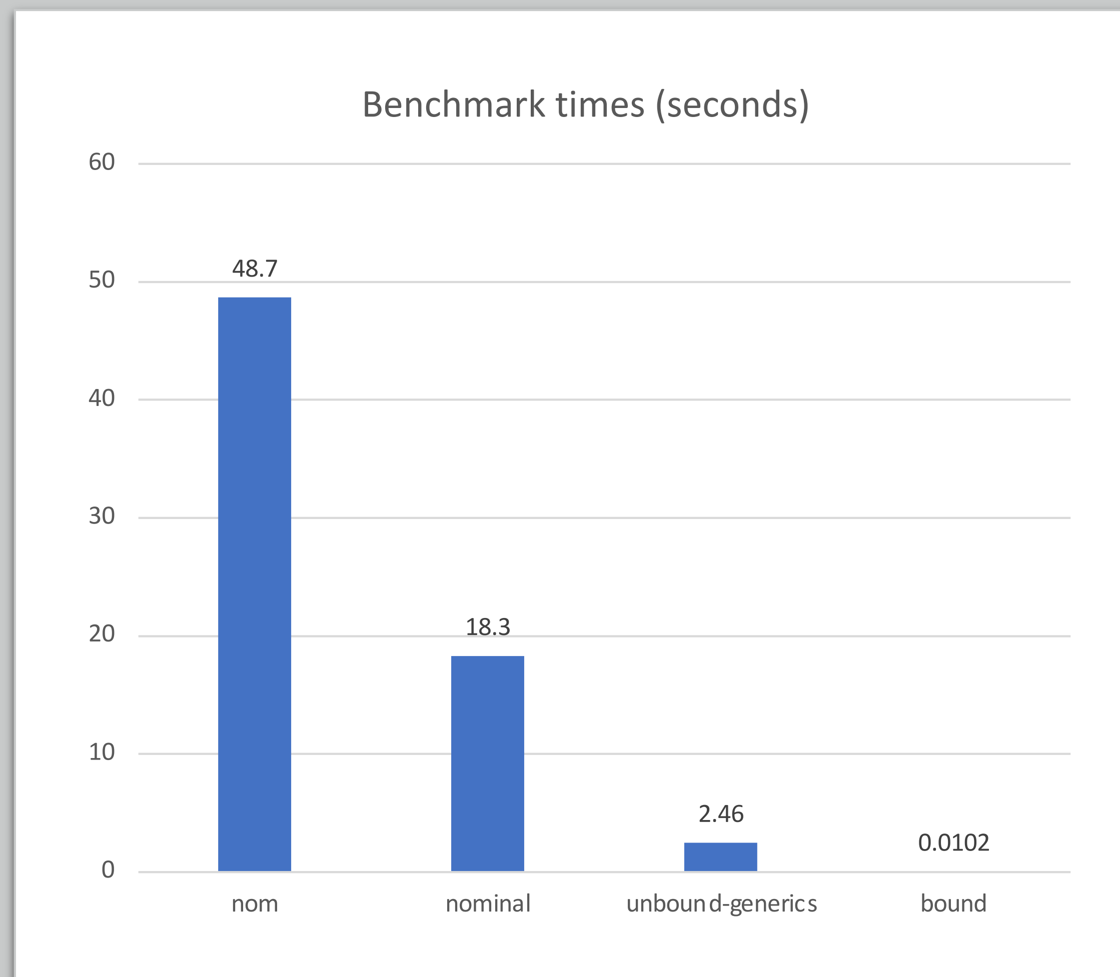
```haskell
import LocallyNameless.UnboundGenerics

data Exp = Var (Name Exp)                    -- x
         | Lam (Bind (Name Exp) Exp)         -- λx.a
         | App Exp Exp                       -- (a b)
            deriving (Show, Generic)

instance Subst Exp Exp where
  isvar (Var x) = Just (SubstName x)
  isvar _ = Nothing
  -- quick definition of substitution
```

Multiple libraries available on hackage!

- Unbound
  (GHC version <= 8.8.3)
- unbound-generics
- bound
- nominal
- nom



GHC 8.8.3, MacBook pro, 2.4 GHz 8-Core Intel Core i9, 64 GB, measured using criterion

Why the difference?

- Different developers?
  - Reimplement uniformly
- Extra features? (pattern binding, annotations, etc.)
  - Only reimplement "core" mechanism
- Magic GHC pragma? `{-# SUMMON SPJ #-}`
  - Study GHC manual
- Different generic programming?
  - Compare with and without
- Different optimizations?
  - Cross-fertilize ideas
- Different binding representations?

- **nom/nominal**
  Name-based binding
- **bound**
  de Bruijn indices
- **Unbound/unbound-generics**
  Locally nameless (mixed)

This work

- Start with three basic implementations
  - Names and simple renaming
  - de Bruijn indices
  - Locally nameless representation
- Create optimized versions
- Use type classes and generics to define a library interface
- Benchmark

Inspiration

λ-calculus cooked four ways

Lennart Augustsson

1 Introduction

This little paper describes how to implement λ-calculus in four different ways. To be precise, it shows how to implement the functions that compute the (β) normal form of an expression.

2005-2006, https://github.com/steshaw/lennart-lambda

Lennart's benchmark

Normal-order reduction of the Church encoding of
6! == sum [1 .. 37] + 17

i.e. 720 == 719

Benchmark statistics

- Number of substitutions required for normalization: **119,697**
- AST depth: 53, binding depth: 25
- Average # of variable occurrences during each beta-reduction: 1.15

Example: Computing normal form with **names**

$$\lambda w.\ \lambda x.\ \underline{(\lambda y.\ w\ y\ (\lambda x.\ w\ y\ (\lambda y.\ w\ y)))\ (x\ x\ x)}$$

substitute (x x x) for y

$$\lambda w.\ \lambda x.\ w\ (x\ x\ x)\ (\underline{\lambda z.}\ w\ (x\ x\ x)\ (\underline{\lambda y.\ w\ y}))$$

rename x *to avoid capture*

stop when y not free

Example: with **de Bruijn** indices

λw. λx. (λy. w y (λx. w y (λy. w y))) (x x x)

λ. λ. (λ. 2 0 (λ. 3 1 (λ. 4 0))) (0 0 0)

substitute (0 0 0) for ***0***

λ. λ. 1 (0 0 0) (λ. 2 (1 1 1) (λ. 3 0))

decrement other variables after reduction

increment (0 0 0) under binders

Example: with **Locally nameless**

$\lambda w.\ \lambda x.\ (\lambda y.\ w\ y\ (\lambda x.\ w\ y\ (\lambda y.\ w\ y)))\ (x\ x\ x)$

name outermost variables to expose beta-reduction

$\underline{\lambda.\ \lambda}.\ (\lambda.\ 2\ 0\ (\lambda.\ 3\ 1\ (\lambda.\ 4\ 0)))\ (0\ 0\ 0)$

$\underline{(\lambda.\ w\ 0\ (\lambda.\ w\ 1\ (\lambda.\ w\ 0)))\ (x\ x\ x)}$

substitute (x x x) for 0

replace names with indices when finished

$w\ (x\ x\ x)\ (\lambda.\ w\ (x\ x\ x)\ (\lambda.\ w\ 0))$

$\lambda.\ \lambda.\ 1\ (0\ 0\ 0)\ (\lambda.\ 2\ (1\ 1\ 1)\ (\lambda.\ 3\ 0))$

Bookkeeping during b {a/x}

- Names: **rename bound variables to avoid capture**
  - calculate free variables of a
  - rename bound variables in b with a "fresh" name
- de Bruijn: **adjust indices**
  - shift free indices in a depending on binding depth
  - decrement indices of b because we lost a binder
- Locally nameless: **exchange names/indices**
  - invariant: a has no "free" indices, only free names
  - exchange happens during traversal, not substitution
  - need to choose "fresh" names

```haskell
data Exp =
    Var Var
  | Lam (Bind Exp)
  | App Exp Exp
```

What does it look like to use these three approaches?

Let's implement normal-order reduction

Normal-order full reduction w/ names

```haskell
nf :: Exp -> Exp
nf (Var x) = Var x
nf (Lam (Bind x e)) =
  -- recurse under binder
  Lam (Bind x (nf e))
nf (App a b) =
  case whnf a of
    Lam (Bind x a0) ->
      nf (subst x b a0)
    a' -> App (nf a') (nf b)
```

```haskell
whnf :: Exp -> Exp
whnf (Var x) = Var x
whnf (Lam b) =
  -- don't recurse
  Lam b
whnf (App a b) =
  case whnf a of
   Lam (Bind x a0) ->
     whnf (subst x b a0)
   a' -> App a' b
       -- normalize head only
```

Normal-order full reduction w/ indices

```
nf :: Exp -> Exp
nf (Var x) = Var x            -- x is an Int (index)
nf (Lam (Bind e)) =           -- no Var stored at binder
  -- recurse under binder
  Lam (Bind (nf e))
nf (App a b) =
  case whnf a of
    Lam (Bind a0) ->
      nf (subst 0 b a0)  -- shift indices during subst
    a' -> App (nf a') (nf b)
```

Reduction w/ locally nameless terms

```
nf :: Exp -> Exp
nf (Var x) = (Var x)          -- invariant, no free indices
nf (Lam (Bind b)) =           -- no Var at binder
  let x  = fresh b            -- find var not free in b
      b' = nf (subst 0 (Var x) b) -- cvt index to name
  in Lam (Bind (close x b'))      -- cvt name to index
nf (App a b) =
  case whnf a of
    Lam (Bind a0) ->
      nf (subst 0 b a0)           -- no shifting (from invariant)
    a' -> App (nf a') (nf b)
```

But which is faster?

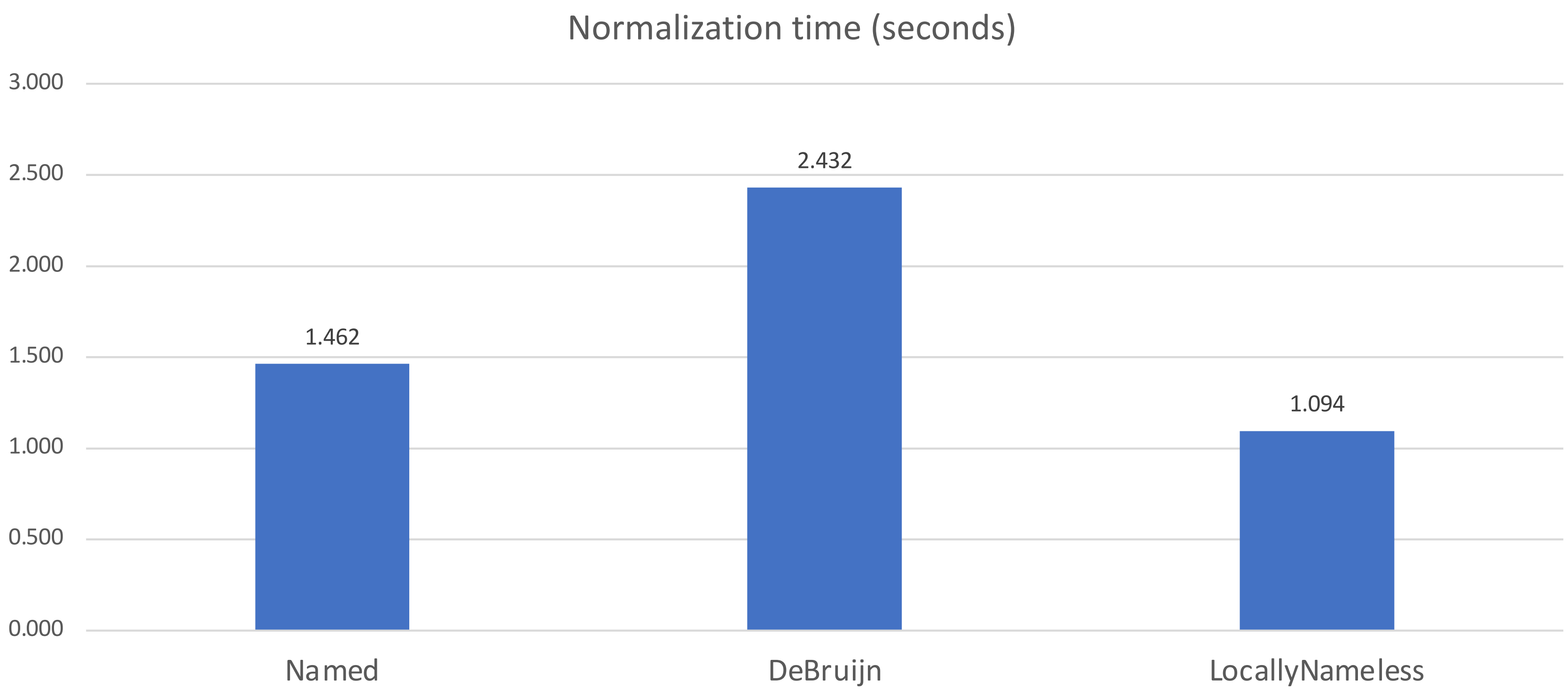
https://github.com/sweirich/lambda-n-ways

Lennart.Simple – adapted from "Lambda-calculus cooked four ways", with bugfix

Lennart.DeBruijn – adapted from "Lambda-calculus cooked four ways"

LocallyNameless.Ott – generated by Ott tool

Benchmark results: basic versions head-to-head



Optimize!

Named.Lazy.SimpleH

DeBruijn.Lazy.Par.B

LocallyNameless.Lazy.Opt

Yaron (Ron) Minsky
@yminsky

Here's a nice PR to OCaml that's fixing some performance bugs in the typechecker.

ocaml/ocaml

#10599 Evaluate signature substitutions lazily

2 comments · 5 reviews · 16 files · +643 -287

stedolan • September 1, 2021 · 10 commits

Evaluate signature substitutions lazily by stedolan · Pull Request #10599 · oca...
When typechecking code that imports large libraries, the compilers spends much of its time evaluating substitutions in signatures (freshening identifiers, renami...
github.com

6:27 AM · Sep 7, 2021 · Twitter Web App

1 Retweet **22** Likes

Can we do better? Yes!

- In all three versions, bookkeeping leads to multiple traversals
- General Idea: generalize traversals and cache info at binders

```
subst :: Sub Exp -> Exp -> Exp
```

- **For Names**
  - Multi-substitution `type Sub a = Map Var a`
  - Cache free vars `data Bind a = Bind VarSet Var a`
- Benefits
  - Find fresh variables quickly
  - Can cut off substitution early if domain doesn't affect free variables
  - Fewer traversals: fuse renaming and normal substitutions

Can we do better? Yes!

- In all three versions, bookkeeping leads to multiple traversals
- General Idea: generalize traversals and cache info at binders

```
subst :: Sub Exp -> Exp -> Exp
```

- **For de Bruijn indices**
  - Multi-substitution
    ```
    data Sub a = Inc Int | Cons a (Sub a)
               | Compose (Sub a) (Sub a)
    ```
  - Delay substitution
    ```
    data Bind a = Bind (Sub a) Var a
    ```
- Benefits
  - Compose substitutions using smart constructors
  - Fewer traversals: multiple indices replaced simultaneously

Can we do better? Yes!

- In all three versions, bookkeeping leads to multiple traversals
- General Idea: generalize traversals and cache info at binders

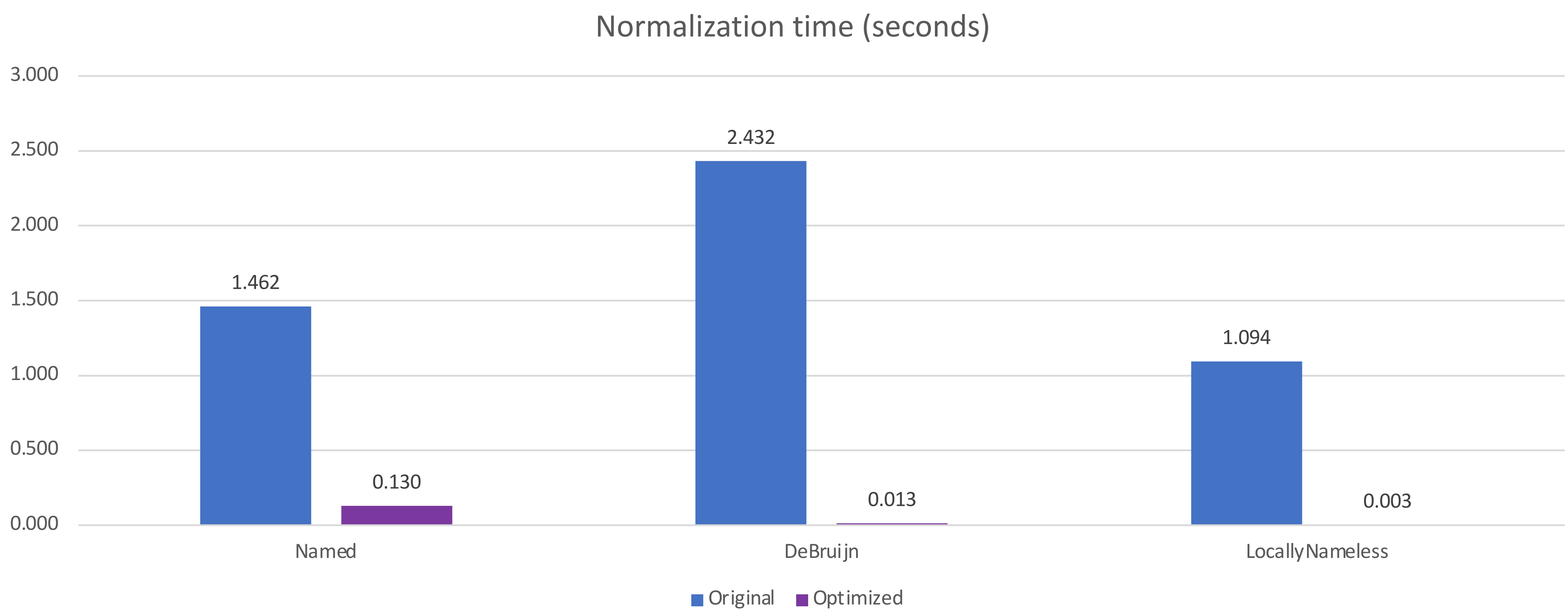
```
subst :: Sub Exp -> Exp -> Exp
```

- **For Locally Nameless**
  - Multi-subst/close
  - Delay last traversal

```
type Sub a = (Int, [Exp])
data Bind a = Bind (Info a) Var a
data Info a = NoInfo | Subst Int [a]
                     | Close Int [IdInt]
```

- Benefits
  - Fewer traversals: multiple indices/names replaced simultaneously

Comparison: Original vs. optimized



```haskell
data Exp =
    Var Var
  | Lam (Bind Exp)
  | App Exp Exp
```

Make it generic!

Named.Lazy.SimpleGH

DeBruijn.Lazy.Par.GB

LocallyNameless.Lazy.GenericInstOpt

Named Client: virtually no code

```
instance VarC Exp where

  var = Var
  isvar (Var v) = Just v
  isvar _ = Nothing

instance FreeVarsC Exp where


instance SubstC Exp Exp where
```

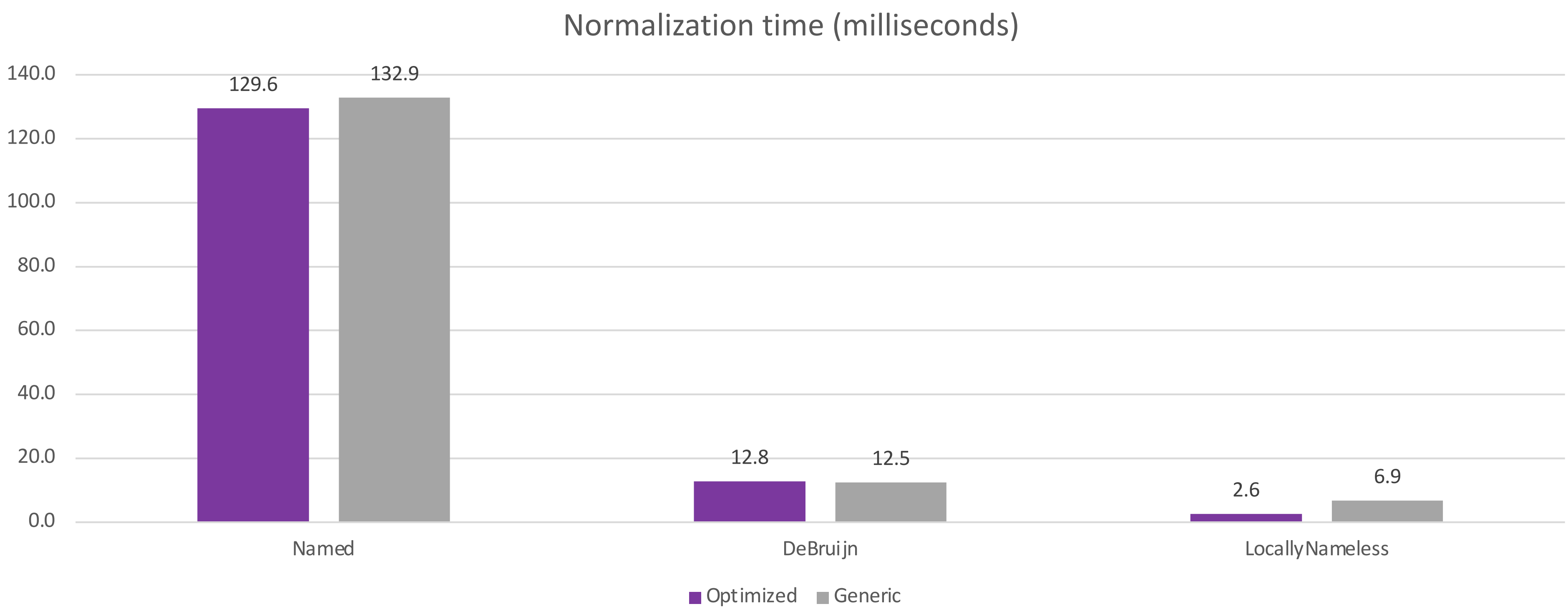
Named Client: virtually no code

```
instance VarC Exp where
  {-# SPECIALIZE instance VarC Exp #-}
  var = Var
  isvar (Var v) = Just v
  isvar _ = Nothing

instance FreeVarsC Exp where
  {-# SPECIALIZE instance FreeVarsC Exp #-}

instance SubstC Exp Exp where
  {-# SPECIALIZE instance SubstC Exp Exp #-}
```

Benchmark: Optimized vs. Generic

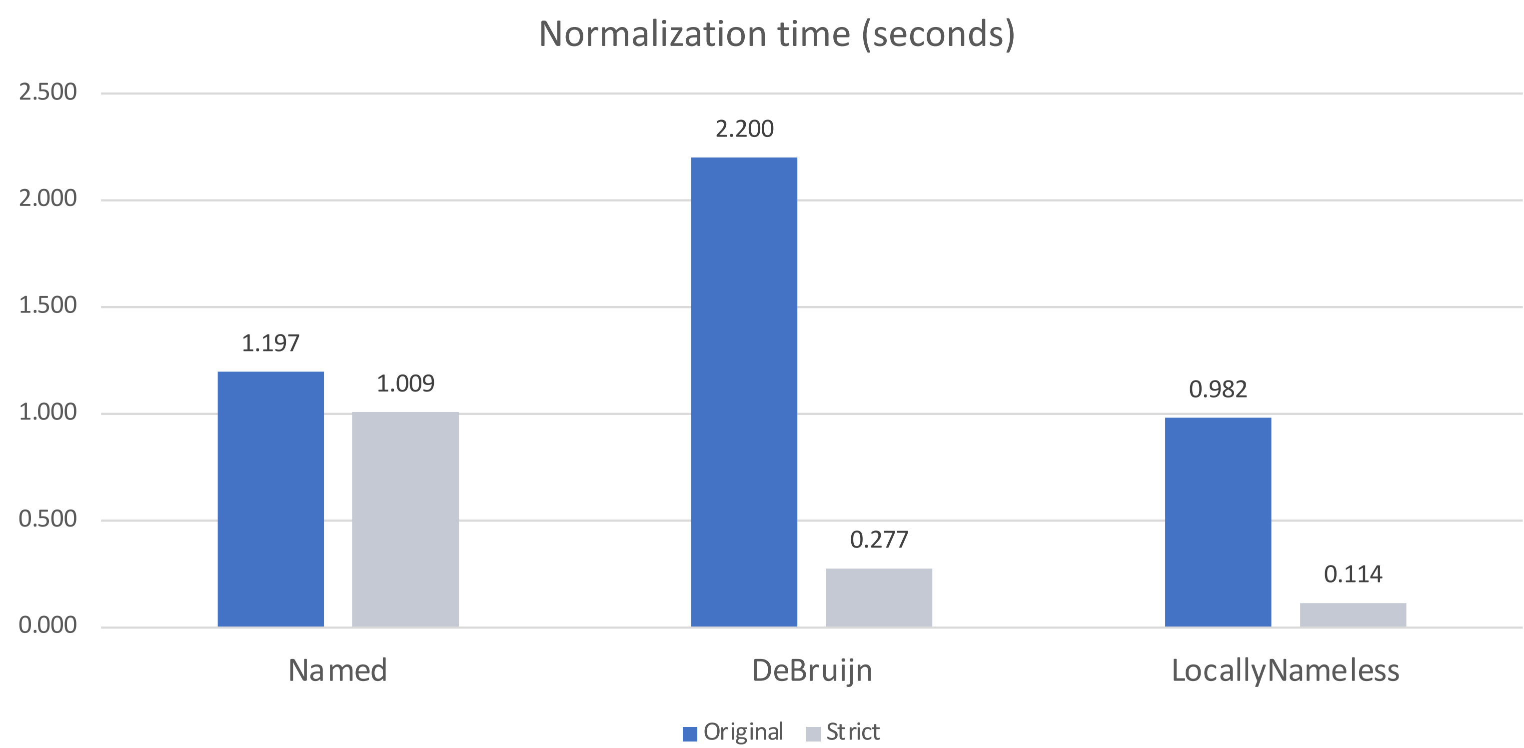


```haskell
data Exp =
    Var Var
  | Lam (Bind Exp)
  | App Exp Exp
```

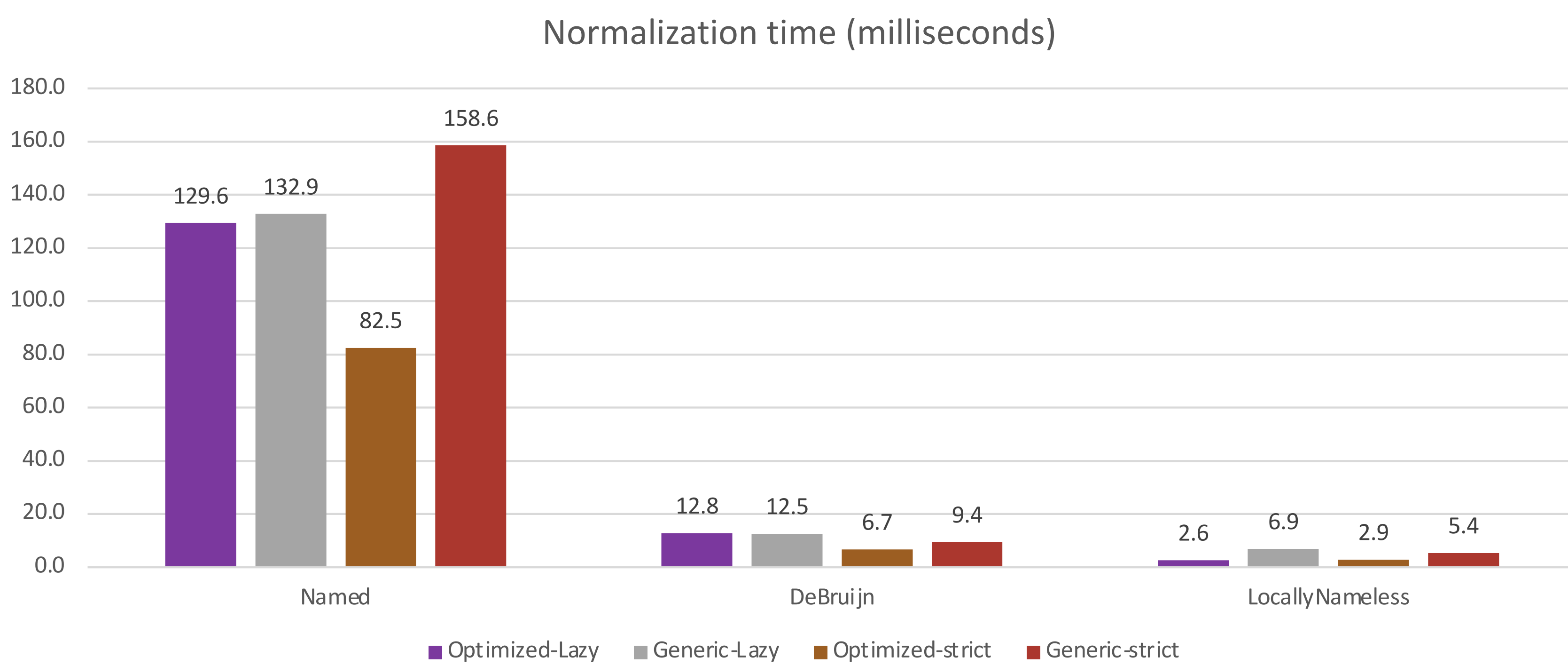
```haskell
data Exp =
    Var {-# UNPACK #-} !Var
  | Lam !(Bind Exp)
  | App !Exp !Exp
```

Coda: strictness?

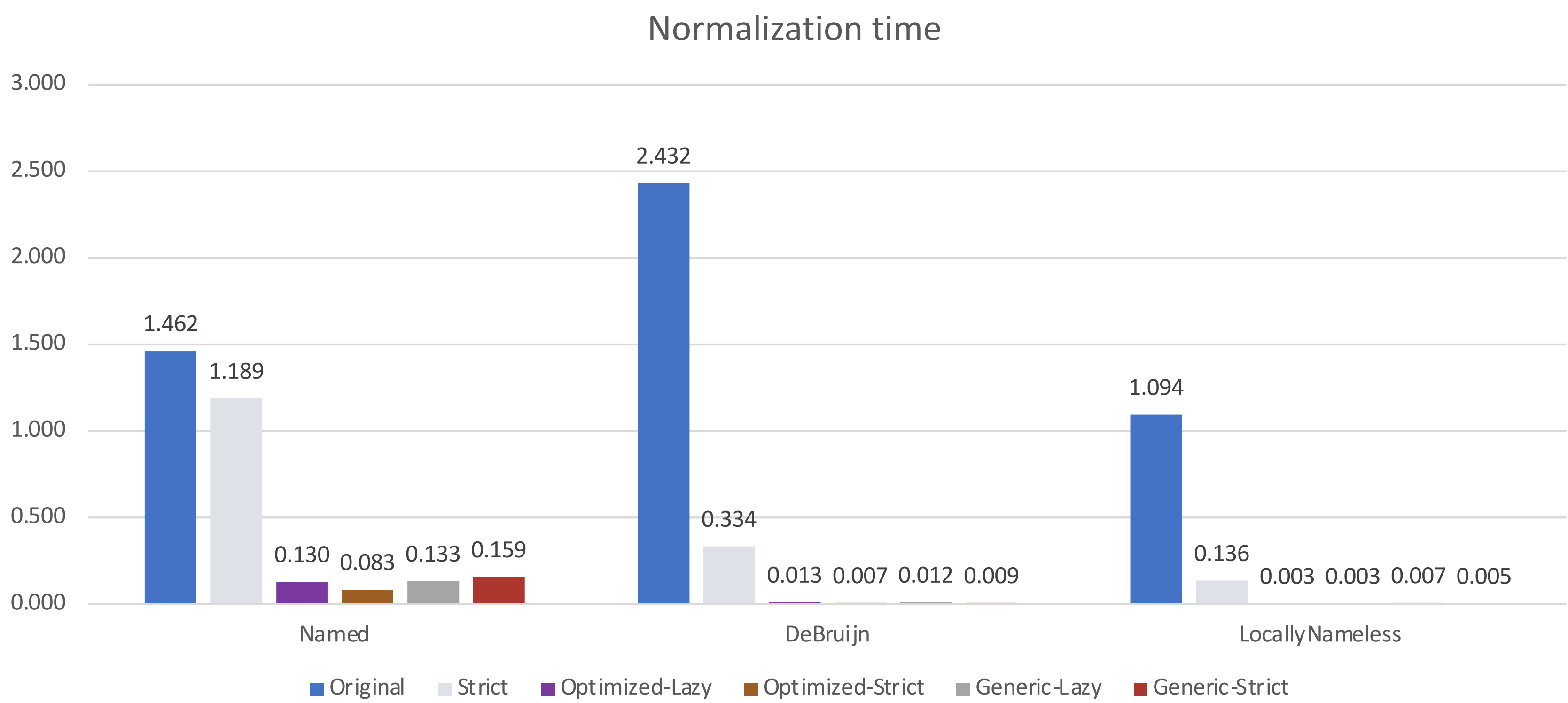
Benchmark results: strictness annotations



Strictness annotations and optimization



Benchmark summary



Conclusions

- This is **one** benchmark, so don't read *too* much into it, but
  - Optimization more significant than representation
  - Generic programming is relatively low runtime cost
- More to do even, in this context
  - de Bruijn optimizations not the same as "bound"
  - locally nameless optimizations not the same as "unbound"
  - Many other alternatives
- Contributions welcome!
  https://github.com/sweirich/lambda-n-ways