# Strongly-typed System F in GHC


- Stephanie Weirich
- University of Pennsylvania


- Chalmers FP seminar
- June 22, 2020

https://github.com/sweirich/challenge/

# Announcement: ICFP 2020

- ONLINE conference, August 23-28

  https://icfp20.sigplan.org/

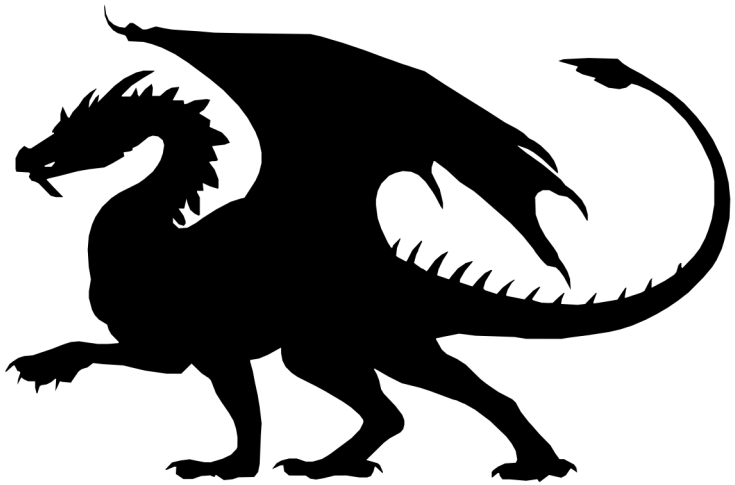
  Accepted paper list is out!
- Now open: **Call for tutorials, panels and discussions**
  - Any topic of interest to FP community, broadly interpreted
  - Submit proposal by July 22nd

# A Challenge Problem

Create a **strongly-typed representation** of **System F** terms, using your favorite programming language

**System F**: typed language with parametric polymorphism

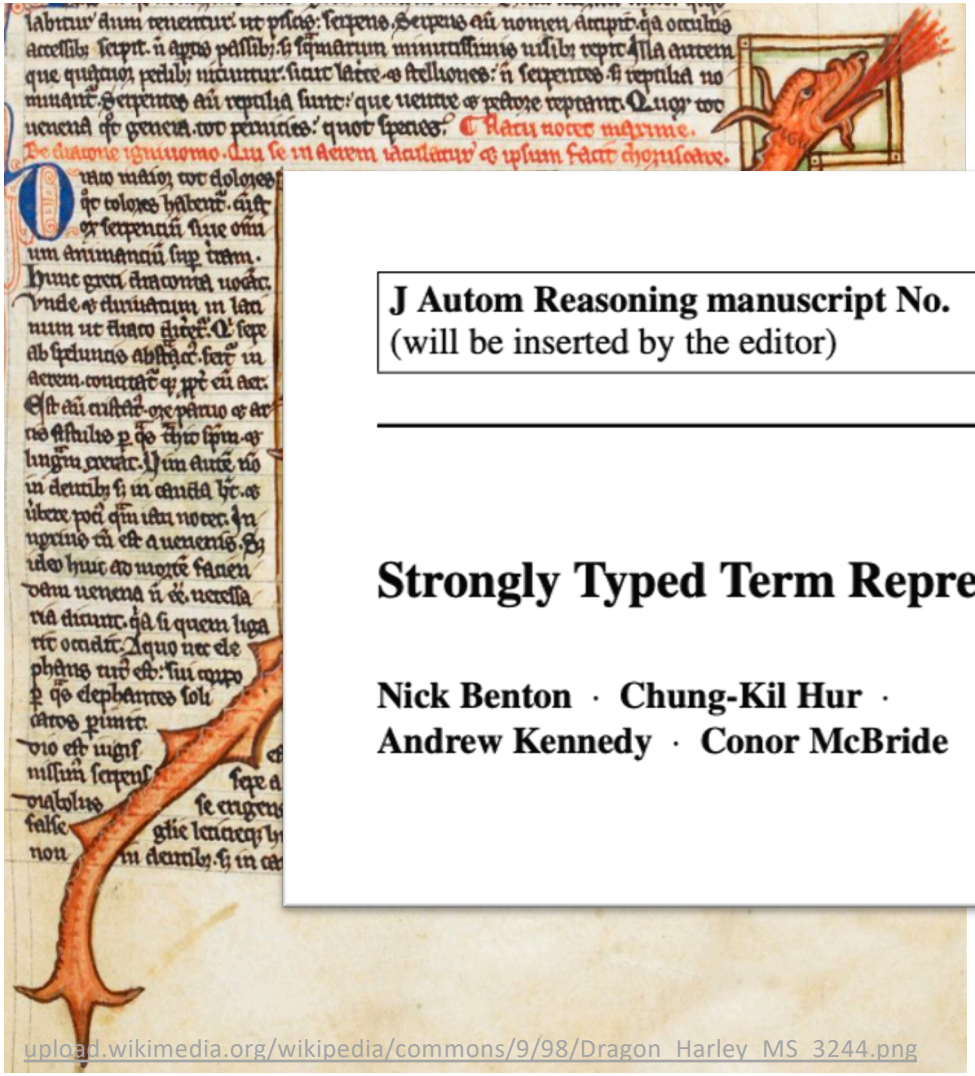
**Strongly-typed representation**: only work with well-typed terms



https://github.com/sweirich/challenge/

# Charting Our Adventure

1. How to implement the Simply-Typed Lambda Calculus (STLC) using de Bruijn indices
   - **reward**: a general interface for substitution
2. The above, but with a strongly-typed representation
   - **reward**: a proof that substitution preserves types
3. The above, but for System F
   - **reward**: example of singletons library in action
   - **caveat**: no fairy tale ending



upload.wikimedia.org/wikipedia/commons/9/98/Dragon_Harley_MS_3244.png

# Our Guide

**J Autom Reasoning manuscript No.**
(will be inserted by the editor)

## Strongly Typed Term Representations in Coq

**Nick Benton · Chung-Kil Hur ·**
**Andrew Kennedy · Conor McBride**

JAR August 2012, Volume 49, Issue 2, pp 141-159

# The Simply-Typed Lambda Calculus (STLC)

$$\frac{x{:}\sigma \in \Gamma}{\Gamma \vdash x{:}\sigma} \text{ (1)} \qquad \frac{c \text{ is a constant of type } T}{\Gamma \vdash c{:}T} \text{ (2)}$$

$$\frac{\Gamma, x{:}\sigma \vdash e{:}\tau}{\Gamma \vdash (\lambda x{:}\sigma.\ e){:}(\sigma \to \tau)} \text{ (3)} \qquad \frac{\Gamma \vdash e_1{:}\sigma \to \tau \quad \Gamma \vdash e_2{:}\sigma}{\Gamma \vdash e_1\ e_2{:}\tau} \text{ (4)}$$

https://en.wikipedia.org/wiki/Simply_typed_lambda_calculus

# Abstract Syntax of STLC

```
data Ty = IntTy | Ty :-> Ty

data Exp where
  IntE :: Int -> Exp          -- constant int, like "3"
  VarE :: Idx -> Exp          -- de Bruijn index (Nat)
  LamE :: Ty -> Exp -> Exp    -- "\(x::ty) -> e"
  AppE :: Exp -> Exp -> Exp   -- "e1 e2"
```

Simple.hs <- Source file if you are following along using the repo

# Abstract Syntax Example

```
\(x :: t1) -> x (\(y :: t2) -> x y)
```

```
LamE t1 (AppE (VarE 0) (LamE t2 (AppE (VarE 1) (VarE 0))))
```

# Small-step reduction, closed terms

```haskell
step :: Exp -> Maybe Exp
step (IntE x)      = Nothing
step (VarE n)      = error "Unbound variable"
step (LamE t e)    = Nothing
step (AppE e1 e2) = Just $ stepApp e1 e2 where
  stepApp :: Exp -> Exp -> Exp
  stepApp (IntE x) e2 = error "Type error"
  stepApp (VarE n) e2 = error "Unbound variable"
  stepApp (LamE t e1)    e2 = ???? -- do a beta redex
  stepApp (AppE e1' e2') e2 = AppE (stepApp e1' e2') e2
```

Simple.hs

# Substitution w/ de Bruijn indices

```
stepApp (LamE t e1) e2    = ???? -- substitute e2 into e1
```

**Simple case**: no lambdas in e1

… replace occurrences of "Var 0" with e2

… if e1 is not closed, decrement *all* other variables (we've removed a binder)

**Towards generality**: what if we need to traverse under a binder?

…replace occurrences of "Var 1" under the binder with e2

…if e1 is not closed, decrement variables >= 2 in e1, but leave 0 alone

…if e2 is not closed, increment free variables in e2 by 1

# Substitution w/ de Bruijn indices

```
stepApp (LamE t e1) e2    = ???? -- substitute e2 into e1
```

**Simple case**: no lambdas in e1

… replace occurrences of "Var 0" with e2

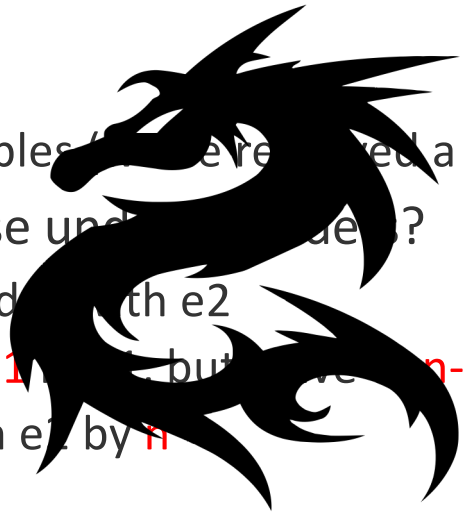
… if e1 is not closed, decrement *all* other variables (because we removed a binder)

**Towards generality**: what if we need to traverse under lambdas?

…replace occurrences of "Var n" under the binder with e2

…if e1 is not closed, decrement variables >= n+1 by 1, but leave n-1 alone

…if e2 is not closed, increment free variables in e2 by n



# Substitution library

```
type Sub                                -- abstract type
applySub  :: Sub -> Idx -> Exp  -- lookup index
lift      :: Sub -> Sub         -- go under binder
```

```
subst :: Sub -> Exp -> Exp
subst s (IntE x)      = IntE x
subst s (VarE x)      = applySub s x
subst s (LamE ty e)   = LamE ty (subst (lift s) e)
subst s (AppE e1 e2) = AppE (subst s e1) (subst s e2)
```

Simple.hs

# Substitution library

```
type Sub a        -- 'a' is the AST type, like Exp
applySub :: SubstDB a => Sub a -> Idx -> a
lift     :: SubstDB a => Sub a -> Sub a

class SubstDB a where
  var   :: Idx -> a              -- var constructor
  subst :: Sub a -> a -> a       -- AST traversal

singleSub :: a -> Sub a       -- create a substitution
```

Subst.hs

# Using the library

```
import Subst

instance SubstDB Exp where
  var = VarE

  subst s (IntE x)      = IntE x
  subst s (VarE x)      = applySub s x
  subst s (LamE ty e)   = LamE ty (subst (lift s) e)
  subst s (AppE e1 e2) = AppE (subst s e1) (subst s e2)
```

Simple.hs

# Small-step reduction, closed terms

```haskell
step :: Exp -> Maybe Exp
step (IntE x)      = Nothing
step (VarE n)      = error "Unbound variable"
step (LamE t e)    = Nothing
step (AppE e1 e2) = Just $ stepApp e1 e2 where
  stepApp :: Exp -> Exp -> Exp
  stepApp (IntE x) e2 = error "Type error"
  stepApp (VarE n) e2 = error "Unbound variable"
  stepApp (LamE t e1)    e2 = subst (singleSub e2) e1
  stepApp (AppE e1' e2') e2 = AppE (stepApp e1' e2') e2
```

Simple.hs

# A strongly-typed AST

Use types to rule out errors

# Strongly-typed AST

```
data Ty = IntTy | Ty :-> Ty

data Exp :: [Ty] -> Ty -> Type where

  IntE :: Int -> Exp g IntTy

  VarE :: Idx g t -> Exp g t

  LamE :: Sing (t1 :: Ty) -> Exp (t1:g) t2
       -> Exp g (t1 :-> t2)

  AppE :: Exp g (t1 :-> t2) -> Exp g t1
       -> Exp g t2
```

The "typing context"
nth Ty in list is type of nth bound variable

Index selects a type from the context (& must be in range)

Add a new type to the context in the body of the lambda (via cons)

I'll come back to this

SimpleTyped.hs

# Strongly-typed substitution library

```
-- A reference to a specific element in the list
data Idx :: [k] -> k -> Type where
  Z :: Idx (t:g) t
  S :: Idx g t -> Idx (u:g) t

-- Idx has a polymorphic kind
  Idx :: forall k. [k] -> k -> Type
  Exp :: [Ty] -> Ty -> Type
```

SubstTyped.hs

# Small-step reduction, closed terms

```
step :: Exp '[] t -> Maybe (Exp '[] t)
step (IntE x)      = Nothing
step (VarE n)      = case n of {}   -- impossible
step (LamE t e)    = Nothing
step (AppE e1 e2) = Just $ stepApp e1 e2 where
  stepApp :: Exp '[] (t1 :-> t2) -> Exp '[] t1 -> Exp '[] t2
  -- stepApp (IntE x) e2 = error "Type error"
  stepApp (VarE n) e2        = case n of {} -- impossible
  stepApp (LamE t e1) e2     = subst (singleSub e2) e1
  stepApp (AppE e1' e2') e2 = AppE (stepApp e1' e2') e2
```

SimpleTyped.hs

# Strongly-typed substitution library

```
-- Substitution applies to indices in context g1
-- and produces terms in context g2
type Sub (a::[k]->k->Type) (g1 :: [k]) (g2 :: [k])

class SubstDB (a :: [k] -> k -> Type) where
  var   :: Idx g t -> a g t
  subst :: Sub a g1 g2 -> a g1 t -> a g2 t

singleSub :: a g t -> Sub a (t:g) g
```

Subst.hs

# Strongly-typed substitution

```
instance SubstDB Exp where
  var :: Idx g t -> Exp g t
  var = VarE

  subst :: Sub Exp g1 g2 -> Exp g1 t -> Exp g2 t
  subst s (IntE x)      = IntE x
  subst s (VarE x)      = applySub s x
  subst s (LamE ty e)   = LamE ty (subst (lift s) e)
  subst s (AppE e1 e2) = AppE (subst s e1) (subst s e2)
```

SubstTyped.hs

# From STLC to System F

## Typing rules [ edit ]

The typing rules of System F are those of simply typed lambda calculus with the addition of the following:

$$\frac{\Gamma \vdash M{:}\forall\alpha.\,\sigma}{\Gamma \vdash M\tau{:}\sigma[\tau/\alpha]}\ (1) \qquad \frac{\Gamma \vdash M{:}\sigma}{\Gamma \vdash \lambda\alpha.\,M{:}\forall\alpha.\,\sigma}\ (2)$$

where $\sigma, \tau$ are types and $\alpha$ is a type variable. The first rule is that of application, and the second is that of abstraction.[2]

https://en.wikipedia.org/wiki/System_F

# Strongly-typed System F

```
data Ty = IntTy | Ty :-> Ty | VarTy Idx | PolyTy Ty

data Exp :: [Ty] -> Ty -> Type where

  IntE :: Int -> Exp g IntTy

  VarE :: S.Idx g t -> Exp g t

  LamE :: Sing (t1 :: Ty) -> Exp (t1:g) t2
       -> Exp g (t1 :-> t2)

  AppE :: Exp g (t1 :-> t2) -> Exp g t1
       -> Exp g t2
  (more)
```

"Weakly-typed" index

Same GADT parameters as STLC: Not tracking scoping of type variables

No change to existing STLC data constructors
More constructors to come

SimpleTyped.hs

# Type substitution in types

```
data Ty = IntTy | Ty :-> Ty | VarTy Idx | PolyTy Ty

instance SubstDB Ty where
  var = VarTy
  subst s IntTy         = IntTy
  subst s (t1 :-> t2) = subst s t1 :-> subst s t2
  subst s (VarTy x)     = applySub s x
  subst s (PolyTy t)  = PolyTy (subst (lift s) t)
```

Poly.hs

# Type substitution in types

```
$(singletons [d|

   data Ty = IntTy | Ty :-> Ty | VarTy Idx | PolyTy Ty

   instance SubstDB Ty where
     var = VarTy
     subst s IntTy         = IntTy
     subst s (t1 :-> t2) = subst s t1 :-> subst s t2
     subst s (VarTy x)    = applySub s x
     subst s (PolyTy t)  = PolyTy (subst (lift s) t)

   |])
```

Poly.hs

# Generated by Singletons

```
data STy :: Ty -> Type where              -- generated
  SIntTy  :: STy IntTy
  (:%->)  :: STy a  -> STy b -> STy (a :-> b)
  SVarTy  :: Sing n -> STy (VarTy n)
  SPolyTy :: STy a  -> STy (PolyTy a)

type family Sing (a :: k) :: Type        -- in library
type instance Sing (a :: Ty) = STy a     -- generated

exTy :: Sing (IntTy :-> IntTy)     -- example
exTy = SIntTy :%-> SIntTy
```

# Generated by Singletons

```
type family Subst (s :: Sub a) (x :: a) :: a
type instance Subst s IntTy         = IntTy
type instance Subst s (t1 :-> t2) = Subst s t1 :-> Subst s t2
type instance Subst s (VarTy x)    = ApplySubst s x
type instance Subst s (PolyTy t)  = PolyTy (Subst (Lift s) t)

sSubst :: forall a (t1 :: Sub a) (t2 :: a).
     SSubstDB a =>
     Sing t1 -> Sing t2 -> Sing (Subst t1 t2)
sSubst s SIntTy = SIntTy
…
```

# System F terms

```
data Ty = IntTy | Ty :-> Ty | VarTy Idx | PolyTy Ty

data Exp :: [Ty] -> Ty -> Type where
  …

  TyApp :: Exp g (PolyTy t1)
        -> Sing (t2 :: Ty)
        -> Exp g (Subst (SingleSub t2) t1)

  TyLam :: Exp (IncList g) t
        -> Exp g (PolyTy t)
```

Singleton type for Ty, via singletons library

Promoted versions of "subst" and "singleSub", via singletons library

Increment all type variables in the context (not shown)

PolyTyped.hs

# Roar!

```
substTy :: forall g s ty. Sing (s :: Sub Ty) -> Exp g ty
        -> Exp (Map (Subst s) g) (Subst s ty)
substTy s (IntE x)      = IntE x
…  -- other cases
substTy s (TyLam (e :: Exp (IncList g) t))
    = TyLam (substTy (sLift s) e)   -- TYPE ERROR

-- Have: Exp (Map (Subst (Lift s)) (IncList g)) (Subst (Lift s) t)
-- TyLam wants: Exp (IncList (Map (Subst s) g)) (Subst (Lift s) t)
    to produce: Exp (Map (Subst s) g) (Subst s (PolyTy t))
```

PolyTyped.hs

# Need type-level reasoning

- GHC cannot show these two types equal

```
Map (Subst (Lift s)) (IncList g) ~
IncList (Map (Subst s) g)
```

- In Coq or Agda we would *prove* it (and Benton et al. do so)
- Haskell is *not* a proof assistant, so what can we do?

# Haskell is not a proof assistant

- Get type class constraint solver to prove it?
  - Works for simple properties, but not here
- Check the property at runtime?
  - Runtime cost for doing the check, typing context not available
- Solution(?)
  - Axiom implemented by unsafeCoerce, carefully marked in a separate file
  - Dangerous; an invalid axiom could cause GHC to segfault
  - Justified via external means (proof in Coq version, QuickCheck tests, etc.)

```
prop_axiom1 :: Sub Ty -> [Ty] -> Bool
prop_axiom1 s g = map (subst (lift s)) (incList g)
                    == incList (map (subst s) g)
```

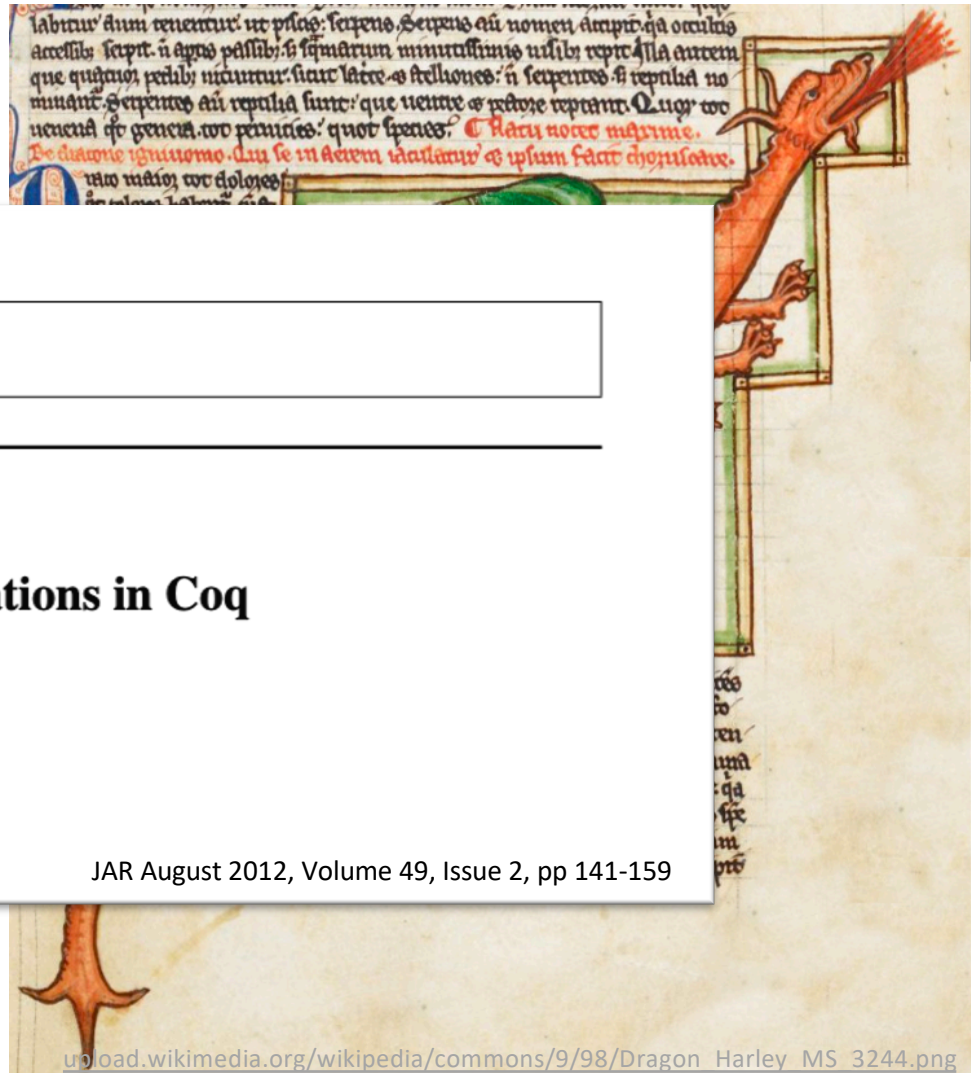
# What did we learn?

J Autom Reasoning manuscript No.
(will be inserted by the editor)

## Strongly Typed Term Representations in Coq

Nick Benton · Chung-Kil Hur ·
Andrew Kennedy · Conor McBride

JAR August 2012, Volume 49, Issue 2, pp 141-159



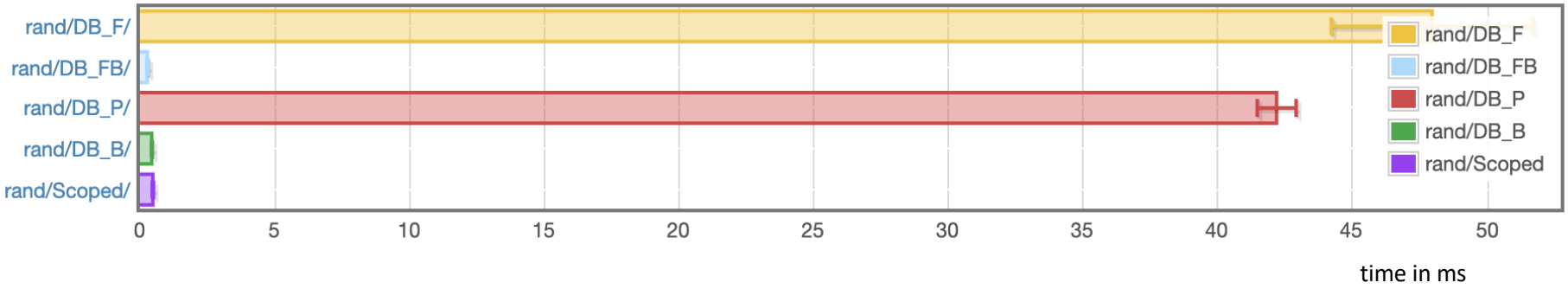
upload.wikimedia.org/wikipedia/commons/9/98/Dragon_Harley_MS_3244.png

# What did we learn?

- Weak internal logic means that there will always be some programs out of reach of the GHC type checker
- Weak internal logic means definitions can be simpler: Coq version must define renaming first, then substitution
- Haskell type classes encourage a general-purpose library (i.e. SubstDB)
- The GHC ecosystem includes profiling/benchmarking tools. How well does this representation work in practice?

# Benchmark – Random terms

Full normalization of  25 randomly generated, closed lambda terms (26-36 subst each)

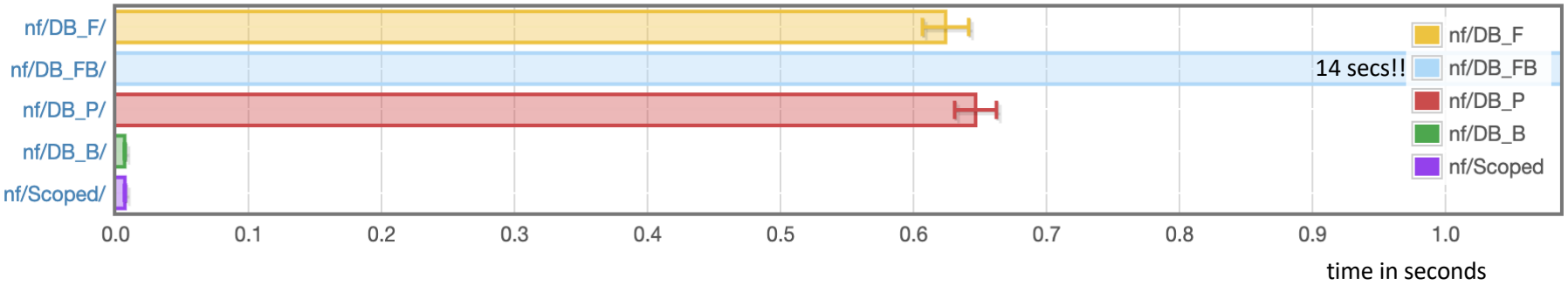



- DB_F:  represent substitutions with functions, as in Benton et al.
- DB_FB: above, but delay substitutions at binders
- DB_P: defunctionalize substitutions
- DB_B: defunctionalize, delay at binders, & "smart" composition
- Scoped: same as DB_B, but types ensure well-scoped terms

https://github.com/sweirich/lennart-lambda

# Benchmark – Pathological term

Full normalization of `factorial 6 == sum [1..37] + 17`, Church encoded (119,697 substs)




- DB_F:  represent substitutions with functions, as in Benton et al.
- DB_FB: above, but delay substitutions at binders
- DB_P: defunctionalize substitutions
- DB_B: defunctionalize, delay at binders, & "smart" composition
- Scoped: same as DB_B, but types ensure well-scoped terms

https://github.com/sweirich/lennart-lambda

# Related Work

- Strongly-typed System F term representations
  - (additional references from Benton et al.)
  - Pfenning and Lee, *Metacircularity in the polymorphic λ-calculus*. Theoretical Computer Science 89 (1991) 137-159
  - Louis-Julien Guillemette, Stefan Monnier. *A type-preserving compiler in Haskell*. ICFP 2008
- Libraries and tools for working generically with de Bruijn indices
  - Edward Kmett. "Bound" library for Haskell
  - Autosubst/Autosubst 2 tool for Coq
  - Allais, Chapman, McBride, McKinna. T*ype-and-scope safe programs and their proofs*. CPP '17
  - Keuchel, Weirich, Schrijvers. *Needle & Knot: Binder Boilerplate Tied Up*. ESOP 2016

# Conclusions

- Strongly-typed System F demonstrates the limit of Dependent Haskell without extending the built-in proof theory
  - No proofs required for STLC
  - No proofs required for *closed* System F terms (step)
  - Type transformations require proofs (substTy, incTy, Cps)
- Important to weigh tradeoffs
  - Strong types can rule out many bugs, but are axioms worth it?
- Strongly-typed representations are compatible with optimization

https://github.com/sweirich/challenge/