Strongly-typed System F in GHC

- Stephanie Weirich
- University of Pennsylvania

- Yow! Lambda Jam Online
- July 24, 2020

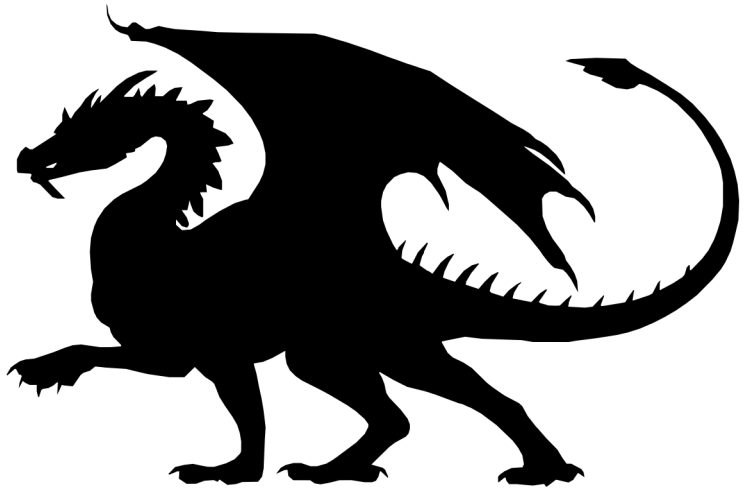
https://github.com/sweirich/challenge/

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

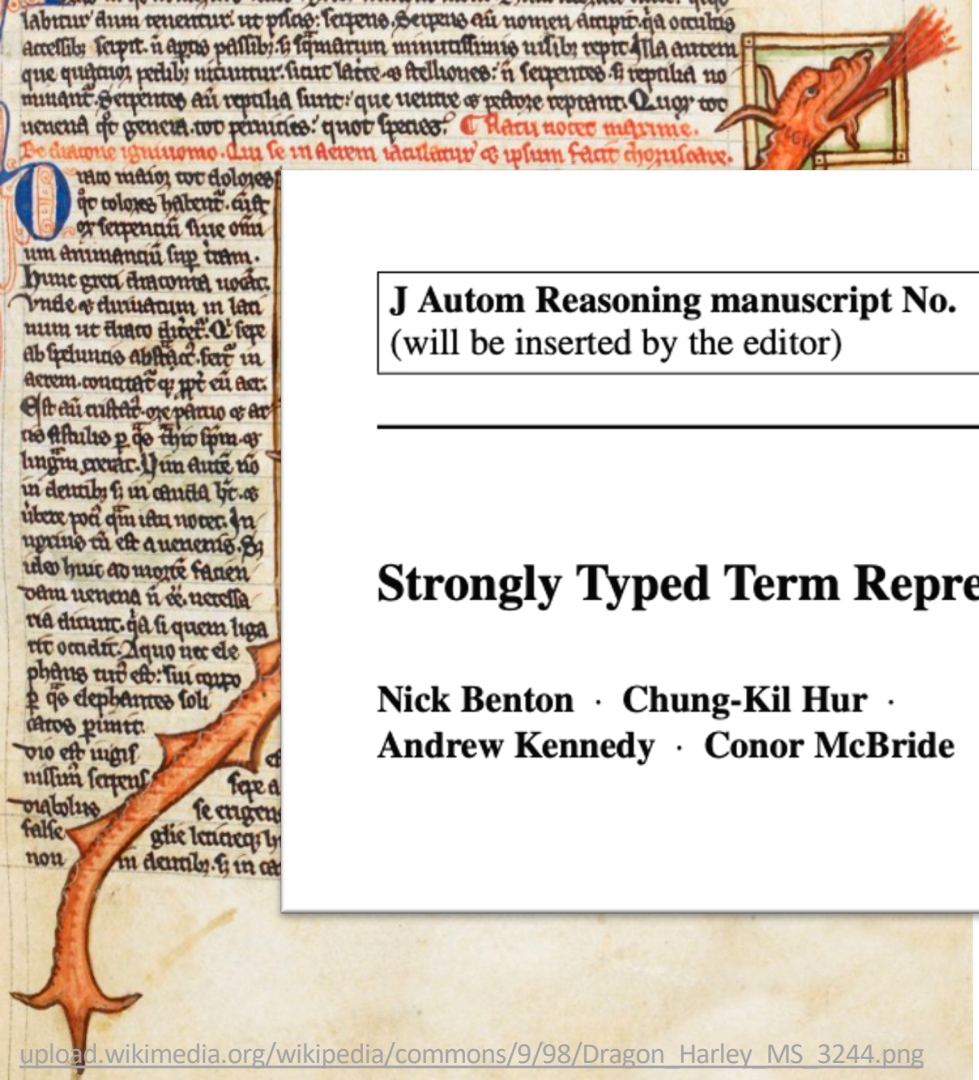


https://github.com/sweirich/challenge/

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

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

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

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 :: Exp -> Exp          -- "\x -> e"
  AppE :: Exp -> Exp -> Exp   -- "e1 e2"
```

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

Abstract Syntax Example

```
\x -> x (\y -> x y)
```

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

Big-step evaluation, closed terms

```
data Val = IntV Int | LamV Exp

eval :: Exp -> Val
eval (IntE x)      = IntV x
eval (VarE _)      = error "Unbound variable"
eval (LamE e)    = LamV e
eval (AppE e1 e2) =
  case eval e1 of
    (IntV _)   -> error "Type error"
    (LamV e1') -> eval ???  -- substitute e2 in e1'
```

Simple.hs

Substitution w/ de Bruijn indices

```
(LamE e1) -> eval ??? -- 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

```
  (LamE e1) -> eval ??? -- 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 n binders?

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

…if e1 is not closed, decrement variables >= n+1 in e1, but leave 0..n-1 alone

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

Substitution library

```
type Sub                              -- abstract type
singleSub :: Exp -> Sub               -- replace 0 with exp
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 e)      = LamE (subst (lift s) e)
subst s (AppE e1 e2) = AppE (subst s e1) (subst s e2)
```

Simple.hs

Substitution library

```
type Sub a                          -- abstract type
singleSub :: a -> Sub a             -- replace 0 with 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
```

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 e)      = LamE (subst (lift s) e)
  subst s (AppE e1 e2) = AppE (subst s e1) (subst s e2)
```

Simple.hs

Big-step evaluation, closed terms

```
data Val = IntV Int | LamV Ty Exp

eval :: Exp -> Val
eval (IntE x)      = IntV x
eval (VarE _)      = error "Unbound variable"
eval (LamE e)      = LamV e
eval (AppE e1 e2) =
  case eval e1 of
    (IntV _)   -> error "Type error"
    (LamV e1') -> eval (subst (singleSub e2) e1)
```

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

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

The type of the expression

Idx 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)

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's kind is an instance of Idx
  Exp :: [Ty] -> Ty -> Type
```

SubstTyped.hs

Big-step evaluation, closed terms

```
data Val :: [Ty] -> Ty -> Type where …

eval :: Exp '[] t -> Val '[] t
eval (IntE x)      = IntV x
-- eval (VarE _)   = error "Unbound variable"
eval (LamE e)      = LamV e
eval (AppE e1 e2) =
  case eval e1 of
    -- (IntV _) -> error "Type error"
    (LamV e1') -> eval (subst (singleSub e2) e1)
```

Simple.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])
singleSub :: a g t -> Sub a (t:g) g

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
```

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 e)      = LamE (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 σ, τ are types and α 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 :: Exp (t1:g) t2
       -> Exp g (t1 :-> t2)

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

"Weakly-typed" index

"Strongly-typed" index, module qualified by S

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

```haskell
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

Promoted (weak)-substitution library

```
$(singletons [d|
   type Sub a                        -- abstract type
   singleSub :: a -> Sub a           -- replace 0 with 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
|])
```

Subst.hs

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

Type substitution in terms

```
substTy :: 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) t1))
    = TyLam (substTy (sLift s) e)   -- TYPE ERROR

-- Have: Exp (Map (Subst (Lift s)) (IncList g)) (Subst (Lift s) t1)
-- TyLam wants: Exp (IncList (Map (Subst s) g)) (Subst (Lift s) t1)
    to produce: Exp (Map (Subst s) g) (Subst s (PolyTy t1))
```

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, need to refactor code
- 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)
```

Conclusions

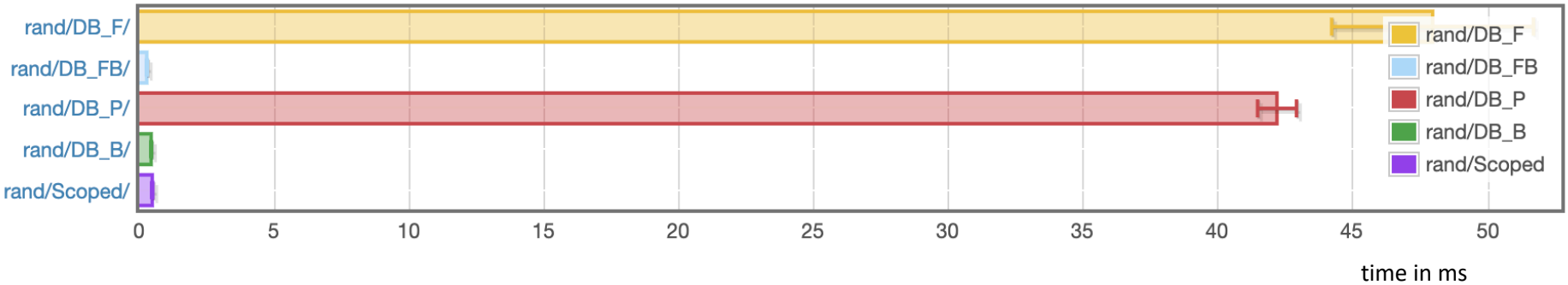
- Strongly-typed System F demonstrates the limit of Dependent Haskell without extending the built-in proof theory
  - No proofs required for STLC
  - No proofs required for *closed* System F terms (eval)
  - Type transformations require proofs (substTy)
- Important to weigh tradeoffs
  - Strong types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

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

Extra slides

Benchmark – Random terms

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

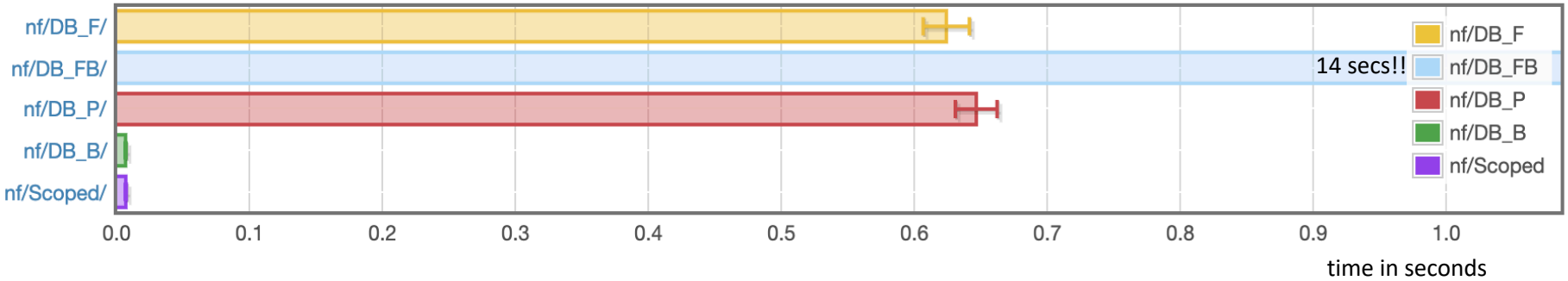


- DB_F: represent substitutions with functions, as in Benton et al.
- DB_FB: above, but delay substitutions at binders
- DB_P: defunctionalize substitutions
- DB_B: defunctionalize, delay at binders, & "smart" composition
- Scoped: same as DB_B, but types ensure well-scoped terms

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

Benchmark – Pathological term

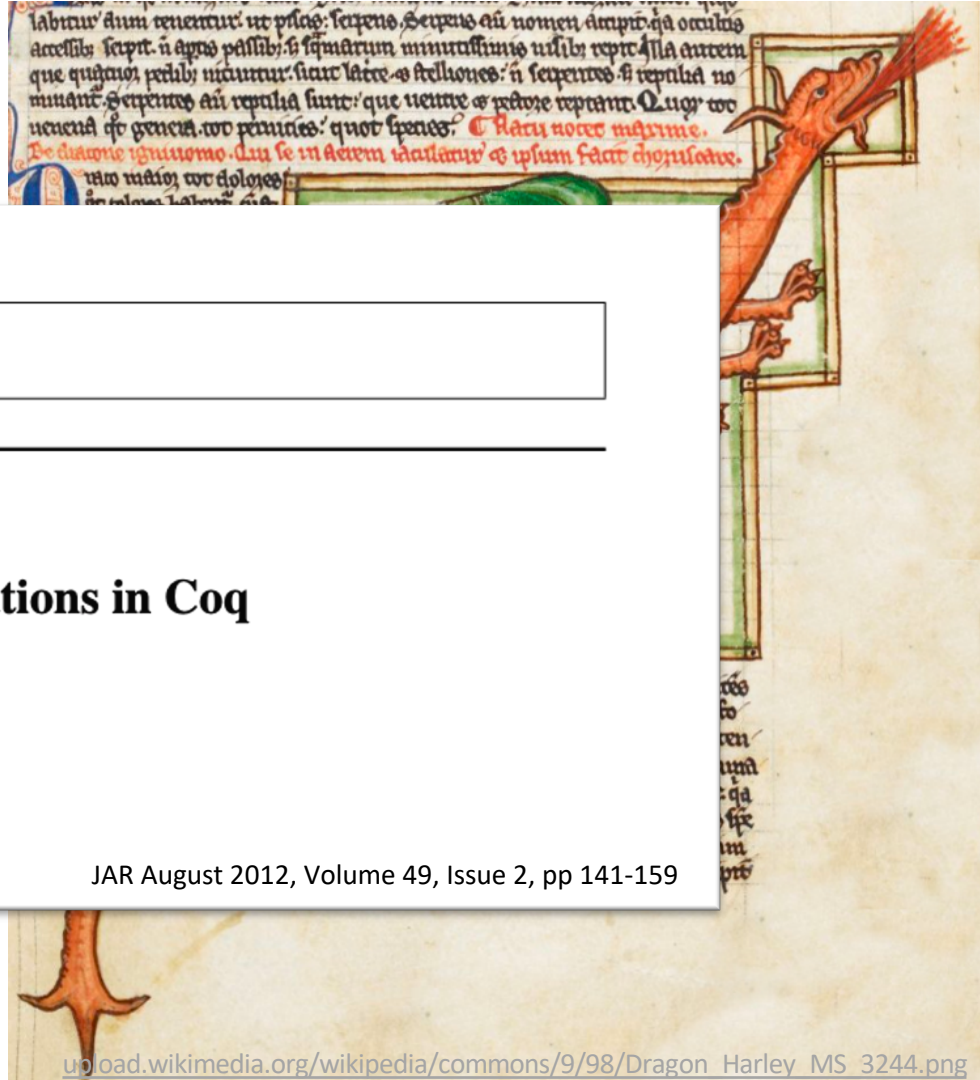
Full normalization of `factorial 6 == sum [1..37] + 17`, Church encoded (119,697 substs)



- 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

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