

Optimizing "Scrap Your Boilerplate" with HERMIT

Michael D. Adams, Andrew Farmer, and José Pedro Magalhães

PL Fest @ Indiana University
October 25, 2013

Hello again, Indiana University!

What have I been doing?

What have I been doing?

- Compiler for the Habit Language (PSU et al.)

What have I been doing?

- Compiler for the Habit Language (PSU et al.)
- Indentation Sensitive Parsing (POPL 2013)
 - Parsec implementation “soon”

What have I been doing?

- Compiler for the Habit Language (PSU et al.)
- Indentation Sensitive Parsing (POPL 2013)
 - Parsec implementation “soon”
- GADT Type Inference

What have I been doing?

- Compiler for the Habit Language (PSU et al.)
- Indentation Sensitive Parsing (POPL 2013)
 - Parsec implementation “soon”
- GADT Type Inference
- Formalizing Macro Hygiene
 - Need help with the Isabel proofs

What have I been doing?

- Compiler for the Habit Language (PSU et al.)
- Indentation Sensitive Parsing (POPL 2013)
 - Parsec implementation “soon”
- GADT Type Inference
- Formalizing Macro Hygiene
 - Need help with the Isabel proofs
- \mathbb{K} Framework (UIUC et al.)
 - Semantics, static analysis, model checking, ...

What have I been doing?

- Compiler for the Habit Language (PSU et al.)
- Indentation Sensitive Parsing (POPL 2013)
 - Parsec implementation “soon”
- GADT Type Inference
- Formalizing Macro Hygiene
 - Need help with the Isabel proofs
- \mathbb{K} Framework (UIUC et al.)
 - Semantics, static analysis, model checking, ...
- Optimizing SYB with HERMIT ...

Scrap Your Boilerplate

```
data AST = ...  
data Lit = ... | Int Int  
  
inc :: Int -> Int  
inc n = n + 1
```

Scrap Your Boilerplate

```
data AST = ...
```

```
data Lit = ... | Int Int
```

```
inc :: Int -> Int
```

```
inc n = n + 1
```

```
incHand/AST (Lam x body) = Lam x (incHand/AST body)
```

Scrap Your Boilerplate

```
data AST = ...
```

```
data Lit = ... | Int Int
```

```
inc :: Int -> Int
```

```
inc n = n + 1
```

```
incHand/AST (Lam x body) = Lam x (incHand/AST body)
```

```
incHand/AST (App e1 e2) = App (incHand/AST e1)  
                           (incHand/AST e2)
```

Scrap Your Boilerplate

```
data AST = ...
```

```
data Lit = ... | Int Int
```

```
inc :: Int -> Int
```

```
inc n = n + 1
```

```
incHand/AST (Lam x body) = Lam x (incHand/AST body)
```

```
incHand/AST (App e1 e2) = App (incHand/AST e1)  
                           (incHand/AST e2)
```

```
incHand/AST (Var x) = Var x
```

Scrap Your Boilerplate

```
data AST = ...
```

```
data Lit = ... | Int Int
```

```
inc :: Int -> Int
```

```
inc n = n + 1
```

```
incHand/AST (Lam x body) = Lam x (incHand/AST body)
```

```
incHand/AST (App e1 e2) = App (incHand/AST e1)  
                        (incHand/AST e2)
```

```
incHand/AST (Var x) = Var x
```

```
incHand/AST (Lit l) =
```

Scrap Your Boilerplate

```
data AST = ...
```

```
data Lit = ... | Int Int
```

```
inc :: Int -> Int
```

```
inc n = n + 1
```

```
incHand/AST (Lam x body) = Lam x (incHand/AST body)
```

```
incHand/AST (App e1 e2) = App (incHand/AST e1)  
                        (incHand/AST e2)
```

```
incHand/AST (Var x) = Var x
```

```
incHand/AST (Lit l) = Lit (incHand/Lit l)
```

Scrap Your Boilerplate

```
data AST = ...
data Lit = ... | Int Int

inc :: Int -> Int
inc n = n + 1

incHand/AST (Lam x body) = Lam x (incHand/AST body)
incHand/AST (App e1 e2)   = App (incHand/AST e1)
                           (incHand/AST e2)

incHand/AST (Var x) = Var x
incHand/AST (Lit l) = Lit (incHand/Lit l)

incHand/Lit (Char c) = Char c
```


Scrap Your Boilerplate

```
data AST = ...
data Lit = ... | Int Int

inc :: Int -> Int
inc n = n + 1

incHand/AST (Lam x body) = Lam x (incHand/AST body)
incHand/AST (App e1 e2)   = App (incHand/AST e1)
                           (incHand/AST e2)

incHand/AST (Var x) = Var x
incHand/AST (Lit l) = Lit (incHand/Lit l)

incHand/Lit (Char c) = Char c
incHand/Lit ...
```

Scrap Your Boilerplate

```
data AST = ...
data Lit = ... | Int Int

inc :: Int -> Int
inc n = n + 1

incHand/AST (Lam x body) = Lam x (incHand/AST body)
incHand/AST (App e1 e2)  = App (incHand/AST e1)
                           (incHand/AST e2)

incHand/AST (Var x) = Var x
incHand/AST (Lit l) = Lit (incHand/Lit l)

incHand/Lit (Char c) = Char c
incHand/Lit ...
incHand/Lit (Int i) = Int (inc i)
```

Language.Haskell.Syntax

- 100+ Constructors
- 30+ Types
 - Expressions
 - Declarations
 - Statements
 - Patterns
 - ...

Scrap Your Boilerplate

```
data AST = ...  
data Lit = ...  
  
inc :: Int -> Int  
inc n = n + 1
```

Scrap Your Boilerplate

```
data AST = ... deriving (Typeable, Data)
data Lit = ... deriving (Typeable, Data)

inc :: Int -> Int
inc n = n + 1
```

Scrap Your Boilerplate

```
data AST = ... deriving (Typeable, Data)
data Lit = ... deriving (Typeable, Data)
```

```
inc :: Int -> Int
inc n = n + 1
```

```
incrementSYB/AST :: AST -> AST
incrementSYB/AST x = everywhere (mkT inc) x
```

Rodriguez Yakushev [2009]:	36x, 52x, and 69x slowdown
Chakravarty et al. [2009]:	45x, 73x, and 230x slowdown
Brown and Sampson [2009]:	4-23x slowdown
Magalhães et al. [2010]:	3x and 20x slowdown
Adams and DuBuisson [2012]:	~10-100x slowdown
Sculthorpe et al. [2013]:	~5x slowdown

Considered a Hard Problem

Optimizing SYB

SYB code can run as fast as hand written code

Optimizations Design Methodology

- Interactive optimization
- Human intuition instead of automated heuristics

Why SYB is Slow

```
inc :: Int -> Int
```

```
inc n = n + 1
```

```
incrementSYB :: [Int] -> [Int]
```

```
incrementSYB x = everywhere (mkT inc) x
```

Why SYB is Slow

```
inc :: Int -> Int
```

```
inc n = n + 1
```

```
incrementSYB :: [Int] -> [Int]
```

```
incrementSYB x = everywhere (mkT inc) x
```

```
everywhere f x = f (gmapT (everywhere f) x)
```

Why SYB is Slow

```
inc :: Int -> Int
```

```
inc n = n + 1
```

```
incrementSYB :: [Int] -> [Int]
```

```
incrementSYB x = everywhere (mkT inc) x
```

```
everywhere f x = f (gmapT (everywhere f) x)
```

```
gmapT f (C x1...xn) = C (f x1) ... (f xn)
```

```
gmapT f [] ↦ []
```

```
gmapT f (x : xs) ↦ f x : f xs
```

Why SYB is Slow

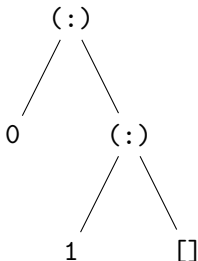
```
inc :: Int -> Int
```

```
inc n = n + 1
```

```
incrementsyb :: [Int] -> [Int]
```

```
incrementsyb x = everywhere (mkT inc) x
```

```
everywhere f x = f (gmapT (everywhere f) x)
```



Why SYB is Slow

```
mkT :: (Typeable a, Typeable b)
      => (b -> b) -> (a -> a)
mkT f = case cast f of
          Nothing -> id
          Just g   -> g
```

```
cast :: (Typeable a, Typeable b) => a -> Maybe b
cast x = r where
  r = if typeOf x == typeOf (fromJust r)
        then Just (unsafeCoerce x)
        else Nothing
```

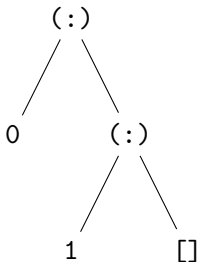
Why SYB is Slow

```
inc :: Int -> Int
```

```
inc n = n + 1
```

```
incrementSYB :: [Int] -> [Int]
```

```
incrementSYB x = everywhere (mkT inc) x
```



Why SYB is Slow

```
everywhere :: (∀b. Data b => b -> b)
            -> (∀a. Data a => a -> a)
gmapT :: (∀b. Data b => b -> b)
       -> (∀a. Data a => a -> a)
mkT :: (Typeable a, Typeable b)
     => (b -> b) -> (a -> a)
cast :: (Typeable a, Typeable b) => a -> Maybe b
```

- Dictionaries
- Runtime type comparison and manipulation
- Intermediary code and data

```
incrementSYB :: [Int] -> [Int]
incrementSYB x = everywhere (mkT inc) x
```



```
incrementSYB :: [Int] -> [Int]  
incrementSYB x = everywhere (mkT inc) x
```

```
incrementHand :: [Int] -> [Int]  
incrementHand [] = []  
incrementHand (x : xs) = inc x : incrementHand xs
```

```
incrementSYB :: [Int] -> [Int]
incrementSYB x = everywhere (mkT inc) x
```

```
incrementHand :: [Int] -> [Int]
incrementHand [] = []
incrementHand (x : xs) = inc x : incrementHand xs
```

Hard to do from just first principles

HERMIT = SSH for GHC

```
$ ghc Increment.hs
```

```
$ ghc Increment.hs -fplugin=HERMIT  
                    -fplugin-opt=HERMIT:main:Main:
```

```
$ ghc Increment.hs -fplugin=HERMIT
                        -fplugin-opt=HERMIT:main:Main:
[1 of 1] Compiling Main ( Increment.hs, Increment.o)
...
```

```
$ ghc Increment.hs -fplugin=HERMIT
                    -fplugin-opt=HERMIT:main:Main:
[1 of 1] Compiling Main ( Increment.hs, Increment.o)
...
module main:Main where
  inc :: Int -> Int
  increment :: [] Int -> [] Int
  ...
```

```
$ ghc Increment.hs -fplugin=HERMIT
                    -fplugin-opt=HERMIT:main:Main:
[1 of 1] Compiling Main ( Increment.hs, Increment.o)
...
module main:Main where
  inc :: Int -> Int
  increment :: [] Int -> [] Int
  ...
hermit<0>
```



```
$ ghc Increment.hs -fplugin=HERMIT
                    -fplugin-opt=HERMIT:main:Main:
[1 of 1] Compiling Main ( Increment.hs, Increment.o)
...
module main:Main where
  inc :: Int -> Int
  increment :: [] Int -> [] Int
  ...
hermit<0> rhs-of 'increment
```

```
$ ghc Increment.hs -fplugin=HERMIT
                    -fplugin-opt=HERMIT:main:Main:
[1 of 1] Compiling Main ( Increment.hs, Increment.o)
...
module main:Main where
  inc :: Int -> Int
  increment :: [] Int -> [] Int
  ...
hermit<0> rhs-of 'increment
λ f x → everywhere (λ a $dData →
    mkT Int a ($p1Data a $dData) $fTypeableInt f)
  [] Int $dData x
```

```
$ ghc Increment.hs -fplugin=HERMIT
                    -fplugin-opt=HERMIT:main:Main:
[1 of 1] Compiling Main ( Increment.hs, Increment.o)
...
module main:Main where
  inc :: Int -> Int
  increment :: [] Int -> [] Int
  ...
hermit<0> rhs-of 'increment
λ f x → everywhere (λ a $dData →
                    mkT Int a ($p1Data a $dData) $fTypeableInt f)
                    [] Int $dData x
hermit<1>
```

```
$ ghc Increment.hs -fplugin=HERMIT
                    -fplugin-opt=HERMIT:main:Main:
[1 of 1] Compiling Main ( Increment.hs, Increment.o)
...
module main:Main where
  inc :: Int -> Int
  increment :: [] Int -> [] Int
  ...
hermit<0> rhs-of 'increment
λ f x → everywhere (λ a $dData →
                    mkT Int a ($p1Data a $dData) $fTypeableInt f)
                    [] Int $dData x
hermit<1> ...
```

```
$ ghc Increment.hs -fplugin=HERMIT
                        -fplugin-opt=HERMIT:main:Main:
[1 of 1] Compiling Main ( Increment.hs, Increment.o)
...
module main:Main where
  inc :: Int -> Int
  increment :: [] Int -> [] Int
  ...
hermit<0> rhs-of 'increment
λ f x → everywhere (λ a $dData →
    mkT Int a ($p1Data a $dData) $fTypeableInt f)
    [] Int $dData x
hermit<1> ...
hermit<127> resume
```

```
$ ghc Increment.hs -fplugin=HERMIT
                    -fplugin-opt=HERMIT:main:Main:
[1 of 1] Compiling Main ( Increment.hs, Increment.o)
...
module main:Main where
  inc :: Int -> Int
  increment :: [] Int -> [] Int
  ...
hermit<0> rhs-of 'increment
λ f x → everywhere (λ a $dData →
                    mkT Int a ($p1Data a $dData) $fTypeableInt f)
                    [] Int $dData x
hermit<1> ...
hermit<127> resume
Linking Increment ...
```

```
$ ghc Increment.hs -fplugin=HERMIT
                        -fplugin-opt=HERMIT:main:Main:
[1 of 1] Compiling Main ( Increment.hs, Increment.o)
...
module main:Main where
  inc :: Int -> Int
  increment :: [] Int -> [] Int
  ...
hermit<0> rhs-of 'increment
λ f x → everywhere (λ a $dData →
    mkT Int a ($p1Data a $dData) $fTypeableInt f)
    [] Int $dData x
hermit<1> ...
hermit<127> resume
Linking Increment ...
$ ./Increment
```

```
incrementSYB :: [Int] -> [Int]
incrementSYB x = everywhere (mkT inc) x
```



```
incrementSYB :: [Int] -> [Int]
incrementSYB = λ x →
  everywhere
    (λ b $dData →
      mkT Int b ($p1Data b $dData) $fTypeableInt inc)
    [Int]
    $dData
    x
```

Optimizing SYB

```
incrementSYB :: [Int] -> [Int]
incrementSYB = λ x →
  mkT Int [Int]
    ($p1Data [Int] $dData)
    $fTypeableInt
    inc
    (gmapT [Int] $dData
      (λ b0 $dData1 →
        everywhere
          (λ b $dData →
            mkT Int b
              ($p1Data b $dData) $fTypeableInt inc)
            b0
            $dData1)
      x)
```

Optimizing SYB

```
incrementSYB :: [Int] -> [Int]
incrementSYB =
  let $dTypeable4 = ...
      $dTypeable5 = ...
  in λ x →
    (case cast (Int -> Int) ([Int] -> [Int]) $dTypeable5 $dTypeable4
     inc of wild
     Nothing → id [Int]
     Just g0 → g0)
    (gmapT [Int] $dData
     (λ b0 $dData1 →
      everywhere
        (λ b $dData →
         mkT Int b
           ($p1Data b $dData)
           $fTypeableInt
           inc)
         b0
         $dData1)
     x)
```

Optimizing SYB

```
incrementSYB :: [Int] -> [Int]
incrementSYB = λ x →
  gmapT [Int] $dData
    (λ b0 $dData1 →
      everywhere
        (λ b $dData →
          mkT Int b
            ($p1Data b $dData) $fTypeableInt inc)
          b0
          $dData1)
  x
```

Optimizing SYB

```
incrementSYB :: [Int] -> [Int]
incrementSYB = λ x →
  case x of wild
    [] → [] Int
    (:) x0 xs0 →
      (:) Int
      (everywhere
        (λ b $dData →
          mkT Int b
            ($p1Data b $dData) $fTypeableInt inc)
        Int
        $fDataInt
        x0)
      (everywhere
        (λ b $dData →
          mkT Int b
            ($p1Data b $dData) $fTypeableInt inc)
        [Int]
        $dData
        xs0)
```

Optimizing SYB

```
incrementSYB :: [Int] -> [Int]
incrementSYB = λ x →
  case x of wild
    [] → [] Int
    (:) x0 xs0 →
      (:) Int
        ((case cast (Int -> Int) (Int -> Int)
              ($fTypeableks (*) (*) (->) Int Int
                ($fTypeableks ((* -> (*)) (*) (->) Int
                  $fTypeable(->)(->) $fTypeableInt)
                  $fTypeableInt)
              ($fTypeableks (*) (*) (->) Int Int
                ($fTypeableks ((* -> (*)) (*) (->) Int
                  $fTypeable(->)(->) ($p1Data Int $fDataInt))
                  ($p1Data Int $fDataInt)))
          inc of wild0
        Nothing → id Int
        Just g0 → g0)
      (gmapT ... )
    (everywhere ... )
```

Optimizing SYB

```
incrementSYB :: [Int] -> [Int]
incrementSYB = λ x →
  case x of wild
    [] → [] Int
    (:) x0 xs0 →
      (:) Int
        (inc
          (gmapT Int $fDataInt
            (λ b0 $dData1 →
              everywhere
                (λ b $dData →
                  mkT Int b
                    ($p1Data b $dData)
                    $fTypeableInt
                    inc)
                  b0
                  $dData1)
            x0))
          (everywhere
            (λ b $dData →
              mkT Int b ($p1Data b $dData) $fTypeableInt inc)
            [Int] $dData xs0)
```

Optimizing SYB

```
incrementSYB :: [Int] -> [Int]
incrementSYB = λ x →
  case x of wild
    [] → [] Int
    (:) x0 xs0 →
      (:) Int
        (inc x0)
        (everywhere
          (λ b $dData →
            mkT Int b
              ($p1Data b $dData) $fTypeableInt inc)
          [Int]
          $dData
          xs0)
```



```
incrementSYB :: [Int] -> [Int]
incrementSYB = λ x →
  case x of wild
    [] → [] Int
    (:) x0 xs0 → (:) Int (inc x0) (incrementSYB xs0)
```

```
incrementHand :: [Int] -> [Int]
incrementHand [] = []
incrementHand (x : xs) = inc x : incrementHand xs
```

- Interactive optimization
 - Hard coded to one traversal
 - Gross manipulations (e.g., `simplify`)
 - Every step is manual

Optimizing SYB Automatically

- Interactive optimization
 - Hard coded to one traversal
 - Gross manipulations (e.g., `simplify`)
 - Every step is manual
- Save commands to a script
- Refactor script
- Load commands from script
- Examine result and repeat

Optimizing SYB Automatically

```
repeat (
  one-td (fold-all >>> trace "!!!! USED MEMOIZED BINDING !!!!!") <+
  any-td (repeat ((apply-rule "map" <+ cast-elim-refl <+ cast-elim-sym-plus <+
    dead-let-elimination <+ let-subst-type '*' <+ let-subst-type 'BOX <+
    eval-fingerprintFingerprints <+ eval-eqWord <+ let-subst-trivial <+
    case-reduce) >>> trace "SIMPLIFYING")) <+
  any-bu (((memo-float-memo-let <+ memo-float-memo-bind <+ memo-float-app <+
    memo-float-arg <+ memo-float-lam <+ memo-float-let <+ memo-float-bind <+
    memo-float-rec-bind <+ memo-float-case <+ memo-float-cast <+ memo-float-alt)
    >>> trace "FLOATING")) <+
  smart-td (when (eliminates-type 'Data <+ eliminates-type 'Typeable <+
    eliminates-type 'Typeable1 <+ eliminates-type 'TypeRep <+
    eliminates-type 'TyCon <+ eliminates-type 'ID <+
    eliminates-type 'Qr <+ eliminates-type 'Fingerprint)
    ((memoize >>> trace "MEMOIZING") <+
    (force ['fingerprintFingerprints, 'eqWord#'] >>> trace "FORCING"))))
```

Custom operators are ~400 lines of Haskell

Algorithm Summary

- 1 Fold memoizations
- 2 Simplify
 - 1 Symmetric and reflexive casts
 - 2 Dead and trivial let binding
- 3 Evaluate primitives
 - 1 `fingerprintFingerprints`
 - 2 `eqWord#`
- 4 Case reduction (OPTIONAL)
- 5 Memoization floating (OPTIONAL)
- 6 At the outermost positions
 - 1 Create memoization, or
 - 2 Eliminate undesirable types

Algorithm Summary

Und τ : Data, Typeable, TypeRep, ID, etc.

$$\frac{\mathbf{ElimUnd} \ e \quad e \rightsquigarrow e'}{e \rightsquigarrow e'} \text{ELIMUND}$$

$$\frac{\vdash e_2 : \tau \quad \mathbf{Und} \ \tau}{\mathbf{ElimUnd} \ (e_1 \ e_2)} \text{ELIMUNDAPP}$$

$$\frac{\vdash e_0 : \tau \quad \mathbf{Und} \ \tau}{\mathbf{ElimUnd} \ (\text{case } e_0 \text{ of } \overrightarrow{p \rightarrow e_i})} \text{ELIMUNDCASE}$$

$$\frac{\vdash e : \tau \quad \mathbf{Und} \ \tau}{\mathbf{ElimUnd} \ (e \triangleright \gamma)} \text{ELIMUNDCAST}$$

Algorithm Summary

- 1 Fold memoizations
- 2 Simplify
 - 1 Symmetric and reflexive casts
 - 2 Dead and trivial let binding
- 3 Evaluate primitives
 - 1 `fingerprintFingerprints`
 - 2 `eqWord#`
- 4 Case reduction (OPTIONAL)
- 5 Memoization floating (OPTIONAL)
- 6 At the outermost positions
 - 1 Create memoization, or
 - 2 Eliminate undesirable types

$$\frac{\mathbf{ElimUnd} \ e \quad e \rightsquigarrow e' \quad \mathbf{Memo} \ e \quad x \notin \mathit{fv}(e')}{e \rightsquigarrow \mathbf{let} \ x : \tau = e' \ \mathbf{in} \ x} \text{MEMOUND}$$

$$\frac{}{\mathbf{Memo} \ x} \text{MEMOUNDVAR}$$

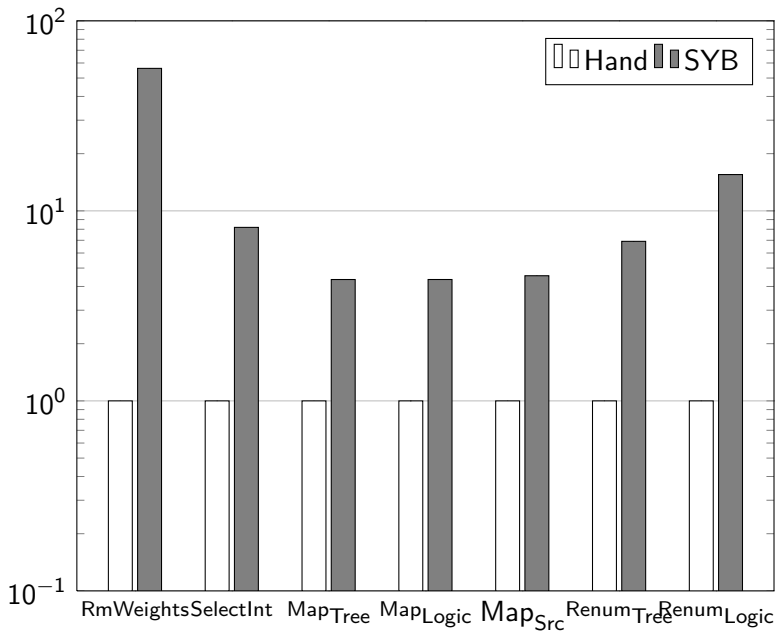
$$\frac{\mathbf{Memo} \ e_1}{\mathbf{Memo} \ (e_1 \ e_2)} \text{MEMOUNDAPP}$$

$$\frac{\mathbf{Memo} \ e_1}{\mathbf{Memo} \ (e_1 \ \tau)} \text{MEMOUNDTYAPP}$$

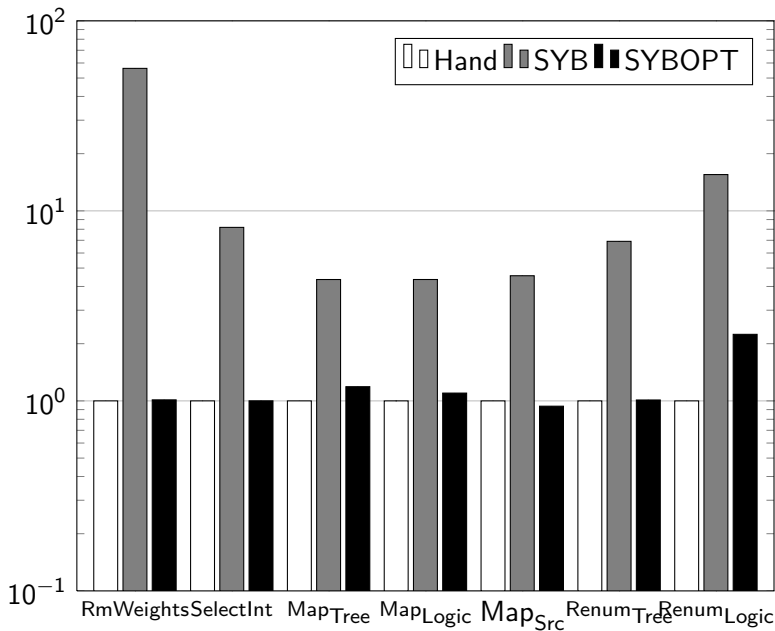
Algorithm Summary

- 1 Fold memoizations
- 2 Simplify
 - 1 Symmetric and reflexive casts
 - 2 Dead and trivial let binding
- 3 Evaluate primitives
 - 1 `fingerprintFingerprints`
 - 2 `eqWord#`
- 4 Case reduction (OPTIONAL)
- 5 Memoization floating (OPTIONAL)
- 6 At the outermost positions
 - 1 Create memoization, or
 - 2 Eliminate undesirable types

Benchmark: Execution Time (Normalized)



Benchmark: Execution Time (Normalized)



- Optimizing SYB is “easy”!
- HERMIT made this optimization easy to implement.
- Optimized SYB can run as fast as handwritten code.

Paper: <http://michaeldadams.org/papers/syb-opt/>

Prototype code: <https://github.com/xich/hermit-syb/>

Presented with HaskellPdfPresenter
As seen on PLWonks!

Michael D. Adams and Thomas M. DuBuisson. Template your boilerplate: Using Template Haskell for efficient generic programming. In *Proceedings of the 2012 ACM SIGPLAN Haskell symposium*, Haskell '12, pages 13–24. ACM, New York, NY, USA, 2012. doi: 10.1145/2364506.2364509

Michael D. Adams and Thomas M. DuBuisson. Template your boilerplate: Using Template Haskell for efficient generic programming. In *Proceedings of the 2012 ACM SIGPLAN Haskell symposium*, Haskell '12, pages 13–24. ACM, New York, NY, USA, 2012. doi: 10.1145/2364506.2364509

- ... but requires working with Template Haskell

Michael D. Adams and Thomas M. DuBuisson. Template your boilerplate: Using Template Haskell for efficient generic programming. In *Proceedings of the 2012 ACM SIGPLAN Haskell symposium*, Haskell '12, pages 13–24. ACM, New York, NY, USA, 2012. doi: 10.1145/2364506.2364509

- ... but requires working with Template Haskell
- HERMIT at ICFP 2012

Michael D. Adams and Thomas M. DuBuisson. Template your boilerplate: Using Template Haskell for efficient generic programming. In *Proceedings of the 2012 ACM SIGPLAN Haskell symposium*, Haskell '12, pages 13–24. ACM, New York, NY, USA, 2012. doi: 10.1145/2364506.2364509

- ... but requires working with Template Haskell
- HERMIT at ICFP 2012
- First prototype implemented in spare time at POPL (~1 week)

HERMIT = SSH for GHC

```
$ ghc Increment.hs
```

```
$ ghc Increment.hs -fplugin=HERMIT  
                    -fplugin-opt=HERMIT:main:Main:
```

```
$ ghc Increment.hs -fplugin=HERMIT
                        -fplugin-opt=HERMIT:main:Main:
[1 of 1] Compiling Main ( Increment.hs, Increment.o)
...
```

```
$ ghc Increment.hs -fplugin=HERMIT
                    -fplugin-opt=HERMIT:main:Main:
[1 of 1] Compiling Main ( Increment.hs, Increment.o)
...
module main:Main where
  inc :: Int -> Int
  increment :: [] Int -> [] Int
  ...
```

```
$ ghc Increment.hs -fplugin=HERMIT
                    -fplugin-opt=HERMIT:main:Main:
[1 of 1] Compiling Main ( Increment.hs, Increment.o)
...
module main:Main where
  inc :: Int -> Int
  increment :: [] Int -> [] Int
  ...
hermit<0>
```



```
$ ghc Increment.hs -fplugin=HERMIT
                    -fplugin-opt=HERMIT:main:Main:
[1 of 1] Compiling Main ( Increment.hs, Increment.o)
...
module main:Main where
  inc :: Int -> Int
  increment :: [] Int -> [] Int
  ...
hermit<0> rhs-of 'increment
```

```
$ ghc Increment.hs -fplugin=HERMIT
                    -fplugin-opt=HERMIT:main:Main:
[1 of 1] Compiling Main ( Increment.hs, Increment.o)
...
module main:Main where
  inc :: Int -> Int
  increment :: [] Int -> [] Int
  ...
hermit<0> rhs-of 'increment
λ f x → everywhere (λ a $dData →
    mkT Int a ($p1Data a $dData) $fTypeableInt f)
  [] Int $dData x
```

```
$ ghc Increment.hs -fplugin=HERMIT
                    -fplugin-opt=HERMIT:main:Main:
[1 of 1] Compiling Main ( Increment.hs, Increment.o)
...
module main:Main where
  inc :: Int -> Int
  increment :: [] Int -> [] Int
  ...
hermit<0> rhs-of 'increment
λ f x → everywhere (λ a $dData →
                    mkT Int a ($p1Data a $dData) $fTypeableInt f)
                    [] Int $dData x
hermit<1>
```

```
$ ghc Increment.hs -fplugin=HERMIT
                    -fplugin-opt=HERMIT:main:Main:
[1 of 1] Compiling Main ( Increment.hs, Increment.o)
...
module main:Main where
  inc :: Int -> Int
  increment :: [] Int -> [] Int
  ...
hermit<0> rhs-of 'increment
λ f x → everywhere (λ a $dData →
                    mkT Int a ($p1Data a $dData) $fTypeableInt f)
                    [] Int $dData x
hermit<1> ...
```

```
$ ghc Increment.hs -fplugin=HERMIT
                        -fplugin-opt=HERMIT:main:Main:
[1 of 1] Compiling Main ( Increment.hs, Increment.o)
...
module main:Main where
  inc :: Int -> Int
  increment :: [] Int -> [] Int
  ...
hermit<0> rhs-of 'increment
λ f x → everywhere (λ a $dData →
    mkT Int a ($p1Data a $dData) $fTypeableInt f)
    [] Int $dData x
hermit<1> ...
hermit<127> resume
```

```
$ ghc Increment.hs -fplugin=HERMIT
                        -fplugin-opt=HERMIT:main:Main:
[1 of 1] Compiling Main ( Increment.hs, Increment.o)
...
module main:Main where
  inc :: Int -> Int
  increment :: [] Int -> [] Int
  ...
hermit<0> rhs-of 'increment
λ f x → everywhere (λ a $dData →
    mkT Int a ($p1Data a $dData) $fTypeableInt f)
    [] Int $dData x
hermit<1> ...
hermit<127> resume
Linking Increment ...
```

```
$ ghc Increment.hs -fplugin=HERMIT
                        -fplugin-opt=HERMIT:main:Main:
[1 of 1] Compiling Main ( Increment.hs, Increment.o)
...
module main:Main where
  inc :: Int -> Int
  increment :: [] Int -> [] Int
  ...
hermit<0> rhs-of 'increment
λ f x → everywhere (λ a $dData →
    mkT Int a ($p1Data a $dData) $fTypeableInt f)
    [] Int $dData x
hermit<1> ...
hermit<127> resume
Linking Increment ...
$
```

GHC Core Syntax

```
\ (f :: GHC.Types.Int -> GHC.Types.Int) ->
  Data.Generics.Schemes.everywhere
    (\ (@ a) ($dData_a2tf :: Data.Data.Data a) ->
      Data.Generics.Aliases.mkT
        @ GHC.Types.Int
        @ a
        (Data.Data.$p1Data @ a $dData_a2tf)
        Data.Typeable.Internal.$fTypeableInt
        f)
    @ [GHC.Types.Int]
    $dData_a2rU
```


HERMIT Clean Syntax

```
λ f →  
  everywhere  
    (λ a $dData →  
      mkT Int a ($p1Data a $dData) $fTypeableInt f)  
    [] Int  
    $dData
```

- GHC API

```
showPpr :: Outputable a =>  
         DynFlags -> a -> String
```

- **Fast** development cycle

- Interactive Optimization

- Backtracking
- Human as Oracle
- “Video Game”

- Step-based optimization

- Critical for finding where optimization first fails

Making Internals Visible

Elimination

FORCEBETA	$(\lambda x : \tau. e_1) e_2$	$\rightsquigarrow \text{let } x : \tau = e_2 \text{ in } e_1$
FORCETYBETA	$(\lambda a : \kappa. e) \tau$	$\rightsquigarrow \text{let } a : \kappa = \tau \text{ in } e$
FORCECASEBETA	$\text{case } K \vec{e}_i \text{ of } \dots K \vec{x}_i : \vec{\tau}_i \rightarrow e_j \dots$	$\rightsquigarrow \text{let } \vec{x}_i : \vec{\tau}_i = \vec{e}_i \text{ in } e_j$
FORCEPUSH	$(e_1 \triangleright \gamma) e_2$	$\rightsquigarrow (e_1 (e_2 \triangleright \text{sym } (\text{nth } 1 \gamma)))) \triangleright (\text{nth } 2 \gamma)$
FORCETYPUSH	$(e \triangleright \gamma) \tau$	$\rightsquigarrow (e \tau) \triangleright (\gamma @ \tau)$
FORCEVAR	x	$\rightsquigarrow e$ [if e is the inlining of x]
FORCELETFLOATAPP	$(\text{let } \vec{x} : \vec{\tau} = \vec{e}_i \text{ in } e_0) u$	$\rightsquigarrow \text{let } \vec{x} : \vec{\tau} = \vec{e}_i \text{ in } e_0 u$
FORCELETFLOATSCR	$\text{case } (\text{let } \vec{x} : \vec{\tau} = \vec{u} \text{ in } e_0) \text{ of } \vec{p}_i \rightarrow \vec{e}_i$	$\rightsquigarrow \text{let } \vec{x} : \vec{\tau} = \vec{u} \text{ in } (\text{case } e_0 \text{ of } \vec{p}_i \rightarrow \vec{e}_i)$
FORCEAPPFUN	$e_1 e_2$	$\rightsquigarrow e'_1 e_2$ [if $e_1 \rightsquigarrow e'_1$]
FORCEAPPTYFUN	$e_1 \tau$	$\rightsquigarrow e'_1 \tau$ [if $e_1 \rightsquigarrow e'_1$]
FORCESCR	$\text{case } e_0 \text{ of } \vec{p}_i \rightarrow \vec{e}_i$	$\rightsquigarrow \text{case } e'_0 \text{ of } \vec{p}_i \rightarrow \vec{e}_i$ [if $e_0 \rightsquigarrow e'_0$]
FORCELETBODY	$\text{let } \vec{x}_i : \vec{\tau}_i = \vec{u}_i \text{ in } e$	$\rightsquigarrow \text{let } \vec{x}_i : \vec{\tau}_i = \vec{u}_i \text{ in } e'$ [if $e_0 \rightsquigarrow e'_0$]
FORCECAST	$e \triangleright \gamma$	$\rightsquigarrow e' \triangleright \gamma$ [if $e \rightsquigarrow e'$]

Simplification

CASTREFL	$e \triangleright \gamma$	$\mapsto e$ if $\vdash^{\text{CO}} \gamma : \tau \sim \tau$
CASTSYM	$e \triangleright \gamma$	$\mapsto e'$ if $e \xrightarrow{\gamma} e'$
DEADLET	let $x : \tau = u$ in e	$\mapsto e$ if $x \notin \text{fv}(e)$ and x is not a memoization
SUBSTSTAR	let $x : \star = \tau$ in e	$\mapsto e[\tau/x]$
SUBSTBOX	let $x : \# = \tau$ in e	$\mapsto e[\tau/x]$
SUBSTVAR	let $x : \tau = x'$ in e	$\mapsto e[x'/x]$
SUBSTLIT	let $x : \tau = l$ in e	$\mapsto e[l/x]$
SUBSTDFUN	let $x : \tau = v \vec{u}$ in e	$\mapsto e[v \vec{u}/x]$ if v is a dictionary constructor

`fingerprintFingerprints e` \rightsquigarrow `[[fingerprintFingerprints e]]`
 if `e` is a value

`fingerprintFingerprints e` \rightsquigarrow `fingerprintFingerprints e'`
 if `e` \rightsquigarrow `e'`

`eqWord# e1 e2` \rightsquigarrow `eqWord# e'1 e2` [if `e1` \rightsquigarrow `e'1`]

`eqWord# e1 e2` \rightsquigarrow `eqWord# e1 e'2` [if `e2` \rightsquigarrow `e'2`]

`eqWord# l1 l2` \rightsquigarrow `True` [if `l1 = l2`]

`eqWord# l1 l2` \rightsquigarrow `False` [if `l1 \neq l2`]

`e` \rightsquigarrow `e'` [if `e` \rightsquigarrow `e'`]

`e1 e2` \rightsquigarrow `e1 e'2` [if `e2` \rightsquigarrow `e'2`]

- Case Reduction

case $K \vec{e}_i$ **of** ... $K \overrightarrow{x_i : \tau_i} \rightarrow e_j \dots \rightsquigarrow$ **let** $\overrightarrow{x_i : \tau_i} = \vec{e}_i$ **in** e_j

- Memoization floating

(Omitted)

Simplification: Cast Symmetry

$$\frac{\vdash^{\text{CO}} \gamma : \tau \sim \tau' \quad \vdash^{\text{CO}} \gamma' : \tau' \sim \tau}{e \triangleright \gamma' \xrightarrow{\gamma} e} \text{CASTSYMCAST}$$

$$\frac{\vdash^{\text{CO}} \gamma : (\tau_1 \rightarrow \tau_2) \sim (\tau_1 \rightarrow \tau'_2) \quad e \xrightarrow{\text{nth}^2 \gamma} e'}{\lambda x : \tau. e \xrightarrow{\gamma} \lambda x : \tau. e'} \text{CASTSYMFUN}$$

$$\frac{e \xrightarrow{\gamma} e'}{\text{let } \overline{x : \tau} = \overrightarrow{e_i} \text{ in } e \xrightarrow{\gamma} \text{let } \overline{x : \tau} = \overrightarrow{e'_i} \text{ in } e'} \text{CASTSYMLET}$$

$$\frac{\overline{} \xrightarrow{\gamma} \overline{} \quad e_i \xrightarrow{\gamma} e'_i}{\text{case } e \text{ of } \overline{p \rightarrow \overrightarrow{e_i}} \xrightarrow{\gamma} \text{case } e \text{ of } \overline{p \rightarrow \overrightarrow{e'_i}}} \text{CASTSYMCASE}$$

- High-Performance Systems-Programming (HASP)
- Generic Programming
 - Template Your Boilerplate
 - Scrap Your Zippers
- Indentation Sensitive Parsing
 - Future Work: LL, PEG, Pretty Printing

- Macro languages
 - Scheme
 - Template Haskell
 - Rust
 - JavaScript
- Unified theory for multiple implementations
- Formally defines “unintended capture”
- Principled approach to extensions
(e.g, `syntax-parameterize`)

Type-Inference Algorithm for GADTs

- Infers more types than “Outside-In”
- Asynchronous type checking
- Avoids “strange” constraints
- Easier to understand and implement

- Extension of work by both Wadler and Curien
- Goal: Practical Language (Cf. λ -calculus vs Scheme)
 - Find Idioms: Disk I/O, Service architecture
- Good model for parallelism
- Data vs Functions
- Theorem Proving
 - Forward vs Backward reasoning
 - Simpler formal foundations
- Makes understanding linear logic easier

- Flow sensitivity in $n \log n$ time
 - More efficient alternative to SSA
- Myers stacks
 - Previous work: $3 \lg n$
 - My (unpublished) work: $2 \lg n$
- Proof of the “Escape Technique” via Galios connections
- Sub-1CFA

Yes

No

Maybe

- Michael D. Adams and Thomas M. DuBuisson. Template your boilerplate: using Template Haskell for efficient generic programming. In *Proceedings of the 2012 Haskell symposium*, Haskell '12, pages 13–24. ACM, 2012. doi: 10.1145/2364506.2364509.
- Neil C.C. Brown and Adam T. Sampson. Alloy: fast generic transformations for Haskell. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell*, pages 105–116. ACM, 2009. doi: 10.1145/1596638.1596652.
- Manuel M. T. Chakravarty, Gabriel C. Ditu, and Roman Leshchinskiy. Instant generics: Fast and easy. Available at <http://www.cse.unsw.edu.au/~chak/papers/instant-generics.pdf>, 2009.
- José Pedro Magalhães, Stefan Holdermans, Johan Jeuring, and Andres Löh. Optimizing generics is easy! In *Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 33–42. ACM, 2010. doi: 10.1145/1706356.1706366.
- Alexey Rodriguez Yakushev. *Towards Getting Generic Programming Ready for Prime Time*. PhD thesis, Utrecht University, 2009.
- Neil Sculthorpe, Nicolas Frisby, and Andy Gill. KURE: A Haskell-embedded strategic programming language with custom closed universes. Under consideration for publication in *J. Functional Programming*, 2013.