Optimizing "Scrap Your Boilerplate" with HERMIT

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

Haskell Implementers Workshop
September 22, 2013

Scrap Your Boilerplate

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

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

Scrap Your Boilerplate

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

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

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

Scrap Your Boilerplate

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

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

inc_Hand/AST (Lam x body) = Lam x (inc_Hand/AST body)
inc_Hand/AST (App e1 e2)  = App (inc_Hand/AST e1)
                                (inc_Hand/AST e2)
```

Scrap Your Boilerplate

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

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

inc_Hand/AST (Lam x body) = Lam x (inc_Hand/AST body)
inc_Hand/AST (App e1 e2)  = App (inc_Hand/AST e1)
                                (inc_Hand/AST e2)
inc_Hand/AST (Var x) = Var x
```

Scrap Your Boilerplate

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

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

inc_Hand/AST (Lam x body) = Lam x (inc_Hand/AST body)
inc_Hand/AST (App e1 e2)  = App (inc_Hand/AST e1)
                                (inc_Hand/AST e2)
inc_Hand/AST (Var x) = Var x
inc_Hand/AST (Lit l) =
```

Scrap Your Boilerplate

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

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

inc_Hand/AST (Lam x body) = Lam x (inc_Hand/AST body)
inc_Hand/AST (App e1 e2)  = App (inc_Hand/AST e1)
                                (inc_Hand/AST e2)
inc_Hand/AST (Var x) = Var x
inc_Hand/AST (Lit l) = Lit (inc_Hand/Lit l)
```

Scrap Your Boilerplate

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

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

inc_Hand/AST (Lam x body) = Lam x (inc_Hand/AST body)
inc_Hand/AST (App e1 e2)  = App (inc_Hand/AST e1)
                                (inc_Hand/AST e2)
inc_Hand/AST (Var x) = Var x
inc_Hand/AST (Lit l) = Lit (inc_Hand/Lit l)

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

Scrap Your Boilerplate

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

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

inc_Hand/AST (Lam x body) = Lam x (inc_Hand/AST body)
inc_Hand/AST (App e1 e2)  = App (inc_Hand/AST e1)
                                (inc_Hand/AST e2)
inc_Hand/AST (Var x) = Var x
inc_Hand/AST (Lit l) = Lit (inc_Hand/Lit l)

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

Scrap Your Boilerplate

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

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

inc_Hand/AST (Lam x body) = Lam x (inc_Hand/AST body)
inc_Hand/AST (App e1 e2)  = App (inc_Hand/AST e1)
                                (inc_Hand/AST e2)
inc_Hand/AST (Var x) = Var x
inc_Hand/AST (Lit l) = Lit (inc_Hand/Lit l)

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

Scrap Your Boilerplate

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

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

SYB Performance

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

Primary Points

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

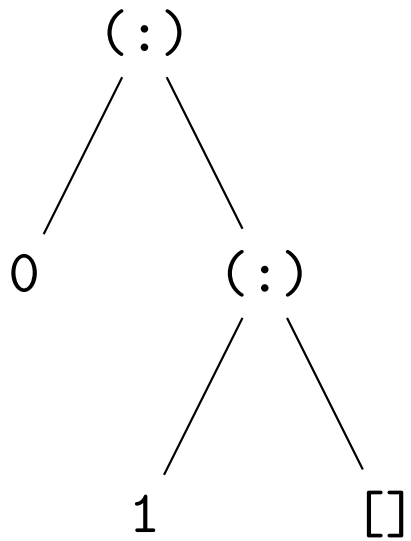
```
inc :: Int -> Int
inc n = n + 1

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

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

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

gmapT f (C x_1...x_n) = C (f x_1) ... (f x_n)

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)
```

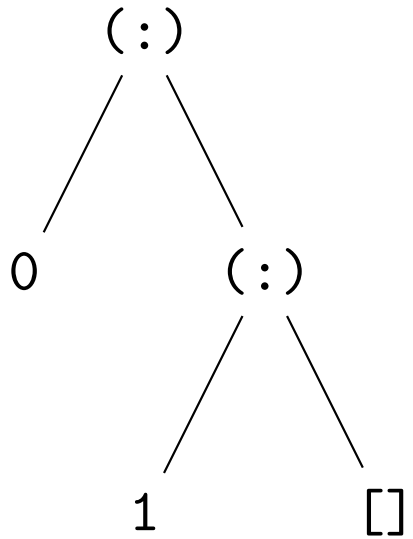


Why SYB is Slow

```
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

Optimizing SYB

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

Optimizing SYB

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

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

Optimizing SYB

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

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

HERMIT

- Interactive optimization
- Human intuition instead of automated heuristics

Optimizing SYB

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

Optimizing SYB

```
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)
```

Optimizing SYB

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

Optimizing SYB

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

Optimizing SYB Automatically

- 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.

$$\frac{\textbf{ElimUnd}\ e \qquad e \dot{\rightsquigarrow} e'}{e \rightarrowtail e'}\ \textsc{ElimUnd}$$

$$\frac{\vdash e_1 : \tau_1 \to \tau_2 \qquad \textbf{Und}\ \tau_1}{\textbf{ElimUnd}\ (e_1\ e_2)}\ \textsc{ElimUndApp}$$

$$\frac{\vdash e_0 : \tau \qquad \textbf{Und}\ \tau}{\textbf{ElimUnd}\ (\textbf{case}\ e_0\ \textbf{of}\ \overrightarrow{p \to e_i})}\ \textsc{ElimUndCase}$$

$$\frac{\vdash e : \tau \qquad \textbf{Und}\ \tau}{\textbf{ElimUnd}\ (e \rhd \gamma)}\ \textsc{ElimUndCast}$$

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

Algorithm Summary

$$\frac{\textbf{ElimUnd}\ e \qquad e \dot{\rightsquigarrow} e' \qquad \textbf{Memo}\ e \qquad x \notin \mathit{fv}\ (e')}{e \rightarrowtail \textbf{let}\ x : \tau = e'\ \textbf{in}\ x}\ \textsc{MemoUnd}$$

$$\frac{}{\textbf{Memo}\ x}\ \textsc{MemoUndVar}$$

$$\frac{\textbf{Memo}\ e_1}{\textbf{Memo}\ (e_1\ e_2)}\ \textsc{MemoUndApp}$$

$$\frac{\textbf{Memo}\ e_1}{\textbf{Memo}\ (e_1\ \tau)}\ \textsc{MemoUndTyApp}$$

Algorithm Summary

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

Algorithm Summary

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

Algorithm Summary

```
incrementSYB :: [Int] -> [Int]
incrementSYB = λ x →
  let memo = 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)
  in memo x
```

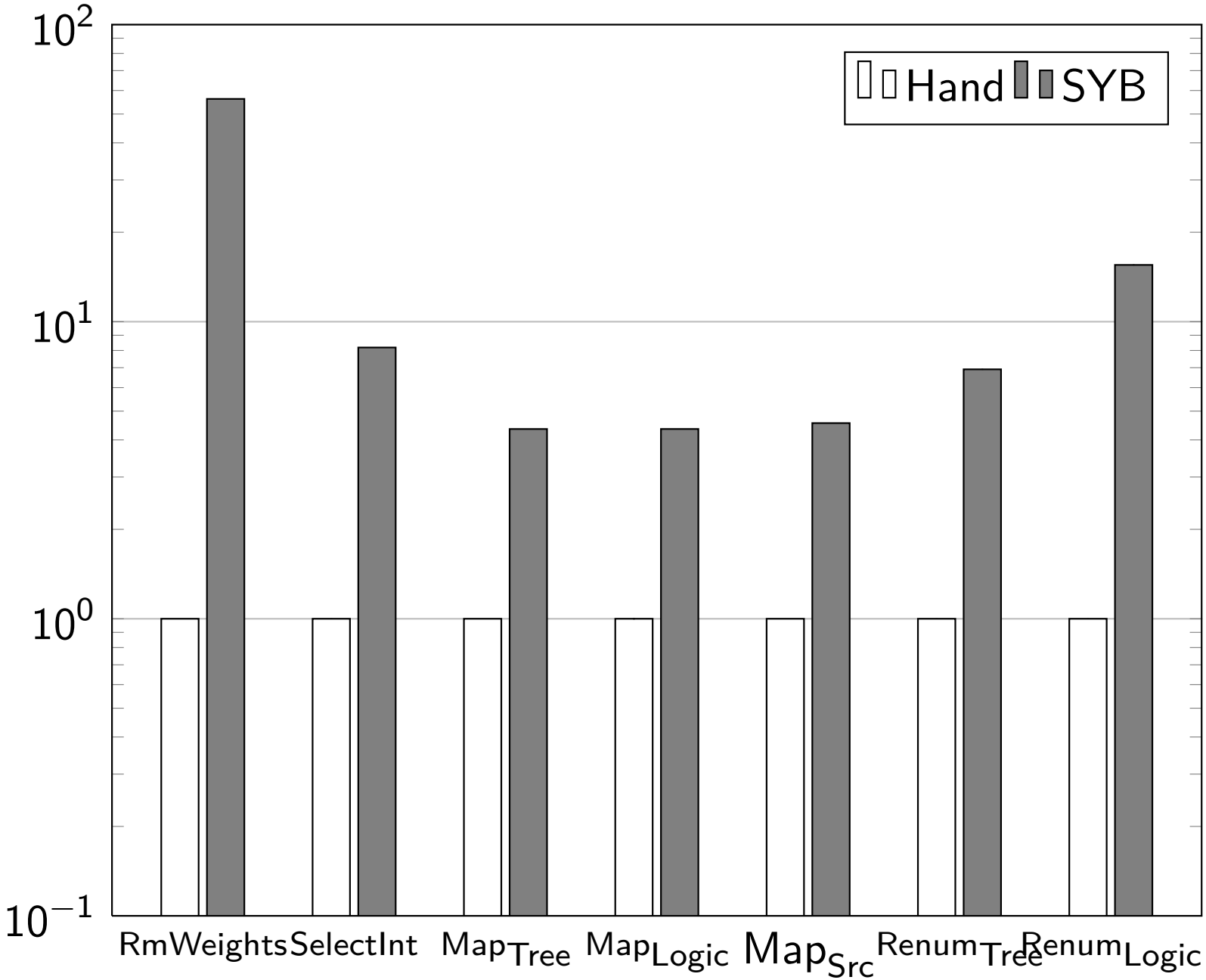
Algorithm Summary

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

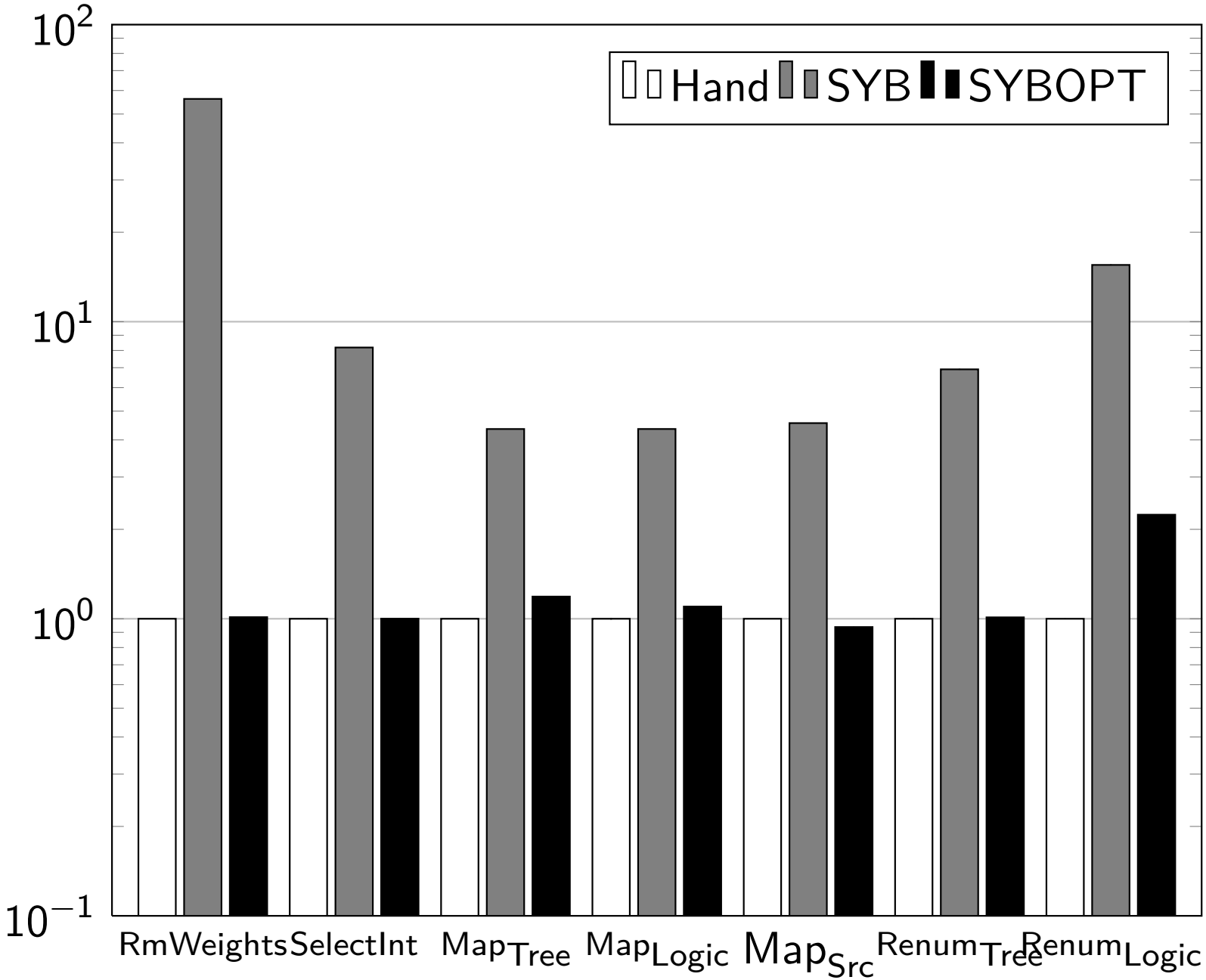
Algorithm Summary

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)



Summary

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

Prototype code in the `optimizations/syb/` folder of HERMIT.

Draft paper available at:

`http://michaeldadams.org/papers/syb-opt/`

Making SYB Faster

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

Making SYB Faster

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

Making SYB Faster

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

Making SYB Faster

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

HERMIT

```
$ ghc Increment.hs
```

HERMIT

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

HERMIT

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

HERMIT

```
$ 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

```
$ 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>
```

HERMIT

```
$ 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
```

HERMIT

```
$ 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

```
$ 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

```
$ 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

```
$ 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
```

HERMIT

```
$ 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 ...
```

HERMIT

```
$ 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
```

HERMIT

- 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	$(\lambda x : \tau.\, e_1)\ e_2$	$\dot{\rightsquigarrow}$ **let** $x : \tau = e_2$ **in** e_1
ForceTyBeta	$(\Lambda a : \kappa.\, e)\ \tau$	$\dot{\rightsquigarrow}$ **let** $a : \kappa = \tau$ **in** e
ForceCaseBeta	**case** $K\ \vec{e_i}$ **of** $\ldots\ K\ \overrightarrow{x_i : \tau_i} \to e_j \ldots$	$\dot{\rightsquigarrow}$ **let** $\overrightarrow{x_i : \tau_i = e_i}$ **in** e_j
ForcePush	$(e_1 \triangleright \gamma)\ e_2$	$\dot{\rightsquigarrow}$ $(e_1\ (e_2 \triangleright \mathbf{sym}\ (\mathbf{nth}\ 1\ \gamma))) \triangleright (\mathbf{nth}\ 2\ \gamma)$
ForceTyPush	$(e \triangleright \gamma)\ \tau$	$\dot{\rightsquigarrow}$ $(e\ \tau) \triangleright (\gamma @ \tau)$
ForceVar	x	$\dot{\rightsquigarrow}$ e [if e is the inlining of x]
ForceLetFloatApp	$\left(\mathbf{let}\ \overrightarrow{x : \tau = e_i}\ \mathbf{in}\ e_0\right)\ u$	$\dot{\rightsquigarrow}$ **let** $\overrightarrow{x : \tau = e_i}$ **in** $e_0\ u$
ForceLetFloatScr	**case** $\left(\mathbf{let}\ \overrightarrow{x : \tau = u}\ \mathbf{in}\ e_0\right)$ **of** $\overrightarrow{p_i \to e_i}$	$\dot{\rightsquigarrow}$ **let** $\overrightarrow{x : \tau = u}$ **in** $\left(\mathbf{case}\ e_0\ \mathbf{of}\ \overrightarrow{p_i \to e_i}\right)$
ForceAppFun	$e_1\ e_2$	$\dot{\rightsquigarrow}$ $e_1'\ e_2$ [if $e_1 \dot{\rightsquigarrow} e_1'$]
ForceAppTyFun	$e_1\ \tau$	$\dot{\rightsquigarrow}$ $e_1'\ \tau$ [if $e_1 \dot{\rightsquigarrow} e_1'$]
ForceScr	**case** e_0 **of** $\overrightarrow{p_i \to e_i}$	$\dot{\rightsquigarrow}$ **case** e_0' **of** $\overrightarrow{p_i \to e_i}$ [if $e_0 \dot{\rightsquigarrow} e_0'$]
ForceLetBody	**let** $\overrightarrow{x_i : \tau_i = u_i}$ **in** e	$\dot{\rightsquigarrow}$ **let** $\overrightarrow{x_i : \tau_i = u_i}$ **in** e' [if $e_0 \dot{\rightsquigarrow} e_0'$]
ForceCast	$e \triangleright \gamma$	$\dot{\rightsquigarrow}$ $e' \triangleright \gamma$ [if $e \dot{\rightsquigarrow} e'$]

Simplification

CastRefl	$e \triangleright \gamma$	$\rightarrowtail e$ if $\vdash^{\mathsf{co}} \gamma : \tau \sim \tau$
CastSym	$e \triangleright \gamma$	$\rightarrowtail e'$ if $e \overset{\gamma}{\hookrightarrow} e'$
DeadLet	$\mathbf{let}\, x : \tau = u\, \mathbf{in}\, e$	$\rightarrowtail e$ if $x \notin fv(e)$ and x is not a memoization
SubstStar	$\mathbf{let}\, x : \star = \tau\, \mathbf{in}\, e$	$\rightarrowtail e[\tau/x]$
SubstBox	$\mathbf{let}\, x : \# = \tau\, \mathbf{in}\, e$	$\rightarrowtail e[\tau/x]$
SubstVar	$\mathbf{let}\, x : \tau = x'\, \mathbf{in}\, e$	$\rightarrowtail e[x'/x]$
SubstLit	$\mathbf{let}\, x : \tau = l\, \mathbf{in}\, e$	$\rightarrowtail e[l/x]$
SubstDFun	$\mathbf{let}\, x : \tau = v\, \vec{u}\, \mathbf{in}\, e$	$\rightarrowtail e[v\, \vec{u}/x]$ if v is a dictionary constructor

Primitives

$$\texttt{fingerprintFingerprints}\ e \quad \dot{\rightsquigarrow}\ [\![\texttt{fingerprintFingerprints}\ e]\!] \quad \text{if } e \text{ is a value}$$

$$\texttt{fingerprintFingerprints}\ e \quad \dot{\rightsquigarrow}\ \texttt{fingerprintFingerprints}\ e' \quad \text{if } e \ \ddot{\rightsquigarrow}\ e'$$

$$\texttt{eqWord\#}\ e_1\ e_2 \quad \dot{\rightsquigarrow}\ \texttt{eqWord\#}\ e_1'\ e_2 \quad [\text{if } e_1 \ \ddot{\rightsquigarrow}\ e_1']$$

$$\texttt{eqWord\#}\ e_1\ e_2 \quad \dot{\rightsquigarrow}\ \texttt{eqWord\#}\ e_1\ e_2' \quad [\text{if } e_2 \ \ddot{\rightsquigarrow}\ e_2']$$

$$\texttt{eqWord\#}\ l_1\ l_2 \quad \dot{\rightsquigarrow}\ \texttt{True} \quad [\text{if } l_1 = l_2]$$

$$\texttt{eqWord\#}\ l_1\ l_2 \quad \dot{\rightsquigarrow}\ \texttt{False} \quad [\text{if } l_1 \neq l_2]$$

$$e \quad \ddot{\rightsquigarrow}\ e' \quad [\text{if } e \ \dot{\rightsquigarrow}\ e']$$

$$e_1\ e_2 \quad \ddot{\rightsquigarrow}\ e_1\ e_2' \quad [\text{if } e_2 \ \ddot{\rightsquigarrow}\ e_2']$$

Optional Simplifications

- Case Reduction

  $$\textbf{case}\ K\ \vec{e_i}\ \textbf{of}\ \ldots K\ \overrightarrow{x_i : \tau_i} \to e_j \ldots \rightsquigarrow \textbf{let}\ \overrightarrow{x_i : \tau_i = e_i}\ \textbf{in}\ e_j$$

- Memoization floating

(Omitted)

Simplification: Cast Symmetry

$$\frac{\vdash^{\mathsf{co}} \gamma : \tau \sim \tau' \qquad \vdash^{\mathsf{co}} \gamma' : \tau' \sim \tau}{e \triangleright \gamma' \overset{\gamma}{\hookrightarrow} e} \textsc{CastSymCast}$$

$$\frac{\vdash^{\mathsf{co}} \gamma : (\tau_1 \to \tau_2) \sim (\tau_1 \to \tau_2') \qquad e \overset{\mathbf{nth}\,2\,\gamma}{\hookrightarrow} e'}{\lambda x : \tau.\, e \overset{\gamma}{\hookrightarrow} \lambda x : \tau.\, e'} \textsc{CastSymFun}$$

$$\frac{e \overset{\gamma}{\hookrightarrow} e'}{\mathbf{let}\ \overrightarrow{x : \tau = e_i}\ \mathbf{in}\ e \overset{\gamma}{\hookrightarrow} \mathbf{let}\ \overrightarrow{x : \tau = e_i}\ \mathbf{in}\ e'} \textsc{CastSymLet}$$

$$\frac{\overrightarrow{e_i \overset{\gamma}{\hookrightarrow} e_i'}}{\mathbf{case}\ e\ \mathbf{of}\ \overrightarrow{p \to e_i} \overset{\gamma}{\hookrightarrow} \mathbf{case}\ e\ \mathbf{of}\ \overrightarrow{p \to e_i'}} \textsc{CastSymCase}$$

Past Research

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

Macro Hygiene and Nominal Logic

- 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

Dual Calculi

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

Control-Flow Analysis

- 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.