Optimizing SYB Is Easy!

Michael D. Adams
Department of Computer Science, University of Illinois at Urbana-Champaign
http://michaeldadams.org/

Andrew Farmer
Information and Telecommunication Technology Center, University of Kansas
afarmer@ittc.ku.edu

José Pedro Magalhães
Department of Computer Science, University of Oxford
jpm@cs.ox.ac.uk

Abstract

The most widely used generic-programming system in the Haskell community, Scrap Your Boilerplate (SYB), also happens to be one of the slowest. Generic traversals in SYB are often an order of magnitude slower than equivalent handwritten, non-generic traversals. Thus while SYB allows the concise expression of many traversals, its use incurs a significant runtime cost. Existing techniques for optimizing other generic-programming systems are not able to eliminate this overhead.

This paper presents an optimization that completely eliminates this cost. Essentially, it is a partial evaluation that takes advantage of domain-specific knowledge about the structure of SYB. It optimizes SYB-style traversals to be as fast as handwritten, non-generic code, and benchmarks show that this optimization improves the speed of SYB-style code by an order of magnitude or more.

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Partial evaluation

Keywords optimization; partial evaluation; datatype-generic programming; Haskell; Scrap Your Boilerplate (SYB); performance

1. Introduction

Scrap Your Boilerplate (SYB) (Lämmel and Peyton Jones 2003, 2004) is one of the oldest and most widely used systems for generic programming in Haskell. It is the most downloaded package for generic programming in the Hackage archive (Industrial Haskell Group 2013). It is easy to use and has strong support from the Glasgow Haskell Compiler (GHC) (GHC Team 2013).

While SYB allows the easy and concise expression of traversals that otherwise require large amounts of handwritten code, it has a serious drawback, namely, poor runtime performance. Our own benchmarks show it to be an order of magnitude slower than handwritten, non-generic code, and this fact is documented many times in the literature (Rodriguez Yakushev 2009; Brown and Sampson 2009; Chakravarty et al. 2009; Magalhães et al. 2010; Adams and DuBuisson 2012; Sculthorpe et al. 2013b).

While attempts have been made in the past to use general-purpose optimizations to improve the performance of SYB, they have met with only moderate success. For example, while setting the compiler's optimizer to be exceptionally aggressive about unfolding and inlining can slightly improve the performance of SYB, doing so can harm the performance of the program as a whole as code may be inlined that should not be (Magalhães et al. 2010).

Nevertheless, SYB-style code exhibits a structure that we can take advantage of in our optimizations. This paper presents a domain-specific optimization that transforms SYB-style code to be as fast as handwritten code. This optimization uses the types of expressions to direct where the inlining process should be more aggressive. In essence, it is a specialized form of supercompilation (Turchin 1979) and partial evaluation (Jones et al. 1993) that uses type information to determine whether an expression should be computed statically at compile time or dynamically at runtime. Using this technique and domain-specific knowledge about the structure of the SYB library and the code that uses it, we show that optimizing SYB-style code can be easily implemented with standard transformations.

This optimization is implemented using HERMIT (Farmer et al. 2012; Sculthorpe et al. 2013a), an interactive optimization system implemented as a GHC plugin. HERMIT makes it easy to quickly develop these sorts of optimizations, and we see our optimization as a prototype which can guide future improvements to the GHC optimizer. The code for this optimization is available at `https://github.com/xich/hermit-syb` and as the `hermit-syb` package on Hackage.

The remainder of this paper is organized as follows. We start with an overview of SYB in Section 2. In Section 3 we show a step-by-step "manual" optimization of an SYB program. This is followed by a formal description of our optimization in Section 4. In Section 5 we discuss an implementation of the optimization for GHC and present benchmarks validating its effectiveness. This is followed in Section 6 by a discussion of the limitations and future work for our system. Finally, we review related work in Section 7, and conclude in Section 8.

2. Overview of SYB

In order to understand why SYB is slow, we must first understand how it works. SYB is a generic-programming system for concisely expressing traversals. For example, suppose we have a type of abstract syntax trees, `AST`, and wish to apply a name mangling function, `mangle`, to every identifier in a given `AST`. Writing this by hand requires a large amount of "boilerplate" code that merely recurs until we get to an identifier where we can apply `mangle`. With SYB, however, we can use the `everywhere` and `mkT` functions to write this traversal simply as `everywhere (mkT mangle)`.

SYB defines many traversals in addition to `everywhere`, and the optimization presented in this paper handles these, but for the sake of simplicity our examples will focus on the `everywhere` traversal. In addition, since traversals over an `AST` type can be

PEPM '14, January 20–21, 2014, San Diego, CA, USA.
Copyright © 2014 ACM 978-1-4503-2619-3/14/01…$15.00.
http://dx.doi.org/10.1145/2543728.2543730

unwieldy, we use the following traversal over slightly simpler types as our running example.

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

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

This traversal applies `inc` to every object in `x` that has type `Int` and thus increments every integer in a list of integers.

We now turn to how `mkT` and `everywhere` work before considering the question of why this SYB-style traversal is slower than an equivalent handwritten traversal.

2.1 Transformations

The `mkT` function applies a transformation `f` to a term `x` if the types are compatible. Otherwise, it behaves as an identity function and simply returns `x`. Its definition relies on the type-safe casting function `cast`, which in turn is defined in terms of the `typeOf` method provided by the `Typeable` class. The implementation of these functions is equivalent to the following although the actual implementation of `mkT` goes through several intermediate helper functions that are not shown here.

```
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
```

The `typeOf` function used in this code returns a value of type `TypeRep` representing the type of its argument. The value of its argument is ignored. The `unsafeCoerce` function has type `∀a b. a -> b` and unconditionally coerces a value of one type to another type. Its use in this code is safe because of the check that the types `a` and `b` are indeed the same.

2.2 Traversals

The `everywhere` function traverses a structure in a bottom-up fashion and is implemented as follows.

```
everywhere :: (∀b. Data b => b -> b)
           -> (∀a. Data a => a -> a)
everywhere f x = f (gmapT (everywhere f) x)
```

It uses `gmapT` to apply `everywhere f` to every subterm of `x`, and afterwards it applies `f` to the result. The `gmapT` function applies a transformation to all the immediate subterms of a given term, and we discuss its implementation in Section 2.3. It does not itself recurse past the first layer of children, but by calling it with `everywhere f` as an argument, the `everywhere` function recurses to all the descendants of `x` in a bottom-up fashion.

2.3 Mapping subterms

The type of `gmapT` is the same as that of `everywhere`. The important difference is that `gmapT` is not recursive, and transforms only the immediate subterms of a term. For any constructor `C` with n arguments, `gmapT` obeys the following equality.

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

The function `gmapT` is a method of the `Data` class, and has a default implementation in terms of the SYB primitive `gfoldl`, which has the following type.

```
gfoldl :: (Data a)
  => (∀d b. Data d => c (d -> b) -> d -> c b)
  -> (∀g. g -> c g) -> a -> c a
```

This is a method of the `Data` class so its implementation is different for every type, but the general structure of such implementations can be seen in the following class instance for lists.

```
instance Data a => Data [a] where
  gfoldl k z []     = z []
  gfoldl k z (x:xs) = z (:) `k` x `k` xs
```

The `gfoldl` function takes three arguments. The first, `k`, combines an argument with the constructor. The second, `z`, is applied to the constructor itself. Finally, the third is the value over which the `gfoldl` method traverses. The implementation always follows the same pattern. For any constructor `C` with n arguments, `gfoldl` obeys the following equality.

```
gfoldl k z (C x₁...xₙ) = z C `k` x₁ ... `k` xₙ
```

While extremely general, `gfoldl` is not easy to use directly. However, generic functions such as `gmapT` that are easier to use can be built in terms of it. Returning to `gmapT`, its default implementation is defined in terms of `gfoldl` as follows.

```
gmapT :: (∀b. Data b => b -> b)
      -> (∀a. Data a => a -> a)
gmapT f x = unID (gfoldl k ID x) where
              k (ID c) y = ID (c (f y))

newtype ID x = ID { unID :: x }
```

Since `gmapT` does not need to take advantage of the type changing ability provided by the `c` type parameter to `gfoldl`, it instantiates `c` to the trivial type `ID`. Aside from wrapping and unwrapping `ID`, `gmapT` operates by using `k` to rebuild the constructor application after applying `f` to each constructor argument and thus obeys the previously given equality for `gmapT`.

2.4 Why SYB is slow

The slow performance of SYB is well documented. Rodriguez Yakushev (2009, Figure 4.9) benchmarked three SYB functions, and found them to be 36, 52, and 69 times slower than handwritten code. Chakravarty et al. (2009) also benchmark SYB on three functions, finding them to be 45, 73, and 230 times slower than handwritten code. Brown and Sampson (2009) developed a new generic-programming library because SYB was too slow and found SYB to be 4 to 23 times slower than their own approach. Magalhães et al. (2010) report SYB performing between 3 and 20 times slower than handwritten code. Adams and DuBuisson (2012) developed an optimized variant of SYB using Template Haskell and report SYB performing between 10 and nearly 100 times slower than handwritten code. Sculthorpe et al. (2013b) benchmark SYB on two generic traversals, finding it to be around 5 times slower than handwritten code. All of these papers conclude that SYB is one of the slowest generic-programming libraries.

After analyzing how SYB works, these results should not be surprising. Consider for example, the runtime behavior of the `increment_SYB` function. When applied to a value of type `[Int]` such as `[0,1]`, it recurses down the structure while applying `mkT inc` to every subterm. In this case, there are five subterms. Three of them are the lists `[0,1]`, `[1]` and `[]`. The remaining two are the `Int` values `0` and `1`. For each subterm, `mkT` attempts to cast `inc` to have a type that is applicable to that subterm. On the lists, it fails to do so, and thus `mkT` returns them unchanged. On the `Int` values, however, the cast succeeds, and thus `mkT` applies `inc` to them. This process involves significant overhead as it uses five dynamic type checks in order to update only two values.

Existing techniques for optimizing other generic-programming libraries are unable to eliminate this overhead in SYB-style code. Since SYB relies heavily on runtime type comparison, the type specializer cannot guide the optimization as it does in the work of Magalhães (2013). Instead, in order to find out if `inc` can be applied to a term, we must inline `mkT`, `cast`, and the `Typeable` methods all the way to the comparison of the type representation computed for the type of a term. If all of those are appropriately inlined, `mkT inc` reduces to either `inc` or `id` depending on whether the types match. However, the GHC inliner (Peyton Jones and Marlow 2002), while often eager to inline small expressions, will not perform as aggressive an inlining as is required here. Coercing GHC to inline aggressively has the side-effect of inlining parts of the code that were not intended to be inlined (Magalhães et al. 2010). Furthermore, because `everywhere` is a recursive function, GHC avoids inlining it in order to ensure termination of the inlining process. Even if GHC would inline recursive definitions, it would have to do so in a way that avoids infinitely inlining nested recursive occurrences. Implementing these optimizations would require fundamental changes to the way the inliner behaves, and their applicability to non-SYB-style code is not clear.

3. Optimizing SYB-style code

In order to gain an intuition for optimizing SYB-style code, we now consider the $\texttt{increment}_{\texttt{SYB}}$ function from Section 2 and how we can manually transform it into non-generic code. Our goal is to reach the following more efficient non-generic implementation that avoids the runtime casts and dictionary dispatches that slow down the code as discussed in Section 2.4.

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

In order to optimize $\texttt{increment}_{\texttt{SYB}}$, we can exploit the fact that, due to the types of $\texttt{increment}_{\texttt{SYB}}$ and `inc`, the concrete types and dictionaries needed by `everywhere` and `mkT` are known at compile time. These can be aggressively inlined, yielding code without any dynamic type checks or runtime casts. In Haskell, type and dictionary arguments are implicit. In order to make them explicit, we represent $\texttt{increment}_{\texttt{SYB}}$ in terms of `Core`, which is the intermediate representation on which GHC does most of its optimizations. The result is the following.

```
incrementSYB :: [Int] -> [Int]
incrementSYB = λ x →
  everywhere
    (λ b0 $dData0 →
       mkT Int b0 ($p1Data b0 $dData0)
         $fTypeableInt inc)
    [Int] $dData x
```

Explicit type arguments are highlighted here in green, and we elide type coercions as they make the code difficult to read. In the following, we also skip many intermediate transformations as the full derivation requires several hundred steps.

In this code, the `$dData` and `$dTypeableInt` variables are `Data` and `Typeable` dictionaries that were previously implicit. The `$p1Data b0 $dData0` expression computes the `Typeable` dictionary corresponding to `$dData0`. We will see more such expressions as we proceed.

Since the dynamic type checks in `mkT` cause this code to be slow, we could try inlining `mkT` immediately. However, we would not have enough information to eliminate these checks if we did so as `b0` and `$dData0` do not yet have values and thus we do not know enough about the arguments to which `mkT` is applied. Instead, in order to get the λ-expression containing `mkT` to a fully applied position, we inline `everywhere`, the function to which it is an argument. This results in the following.

```
incrementSYB :: [Int] -> [Int]
incrementSYB = λ x →
  mkT Int [Int] ($p1Data [Int] $dData) $fTypeableInt inc
    (gmapT [Int] $dData
       (λ b1 $dData1 →
          everywhere
            (λ b0 $dData0 →
               mkT Int b0 ($p1Data b0 $dData0)
                 $fTypeableInt inc)
            b1 $dData1)
       x)
```

The call to `mkT` at the beginning of this code can now be inlined, and this exposes a call to `cast`.

```
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 b1 $dData1 →
               everywhere
                 (λ b0 $dData0 →
                    mkT Int b0 ($p1Data b0 $dData0)
                      $fTypeableInt inc)
                 b1 $dData1)
            x)
```

This code attempts to cast `inc` from type `Int -> Int` to type `[Int] -> [Int]` by using the `cast` function. Inlining `cast` exposes calls to `typeOf` that we can symbolically evaluate. After several more simplification steps, this call to `cast` reduces to `Nothing`, and in turn the `case` statement can be reduced to the identity function. Thus, we have removed one of the runtime type comparisons that slow down this code, and after simplification, the code now looks like the following.

```
incrementSYB :: [Int] -> [Int]
incrementSYB = λ x →
  gmapT [Int] $dData
    (λ b1 $dData1 →
       everywhere
         (λ b0 $dData0 →
            mkT Int b0 ($p1Data b0 $dData0)
              $fTypeableInt inc)
         b1 $dData1)
    x
```

Here again we choose not to inline the λ-expression containing `mkT` since it is not fully applied. Instead we inline `gmapT` and get the following code.

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

```
        (everywhere
           (λ b0 $dData0 →
              mkT Int b0 ($p1Data b0 $dData0)
                $fTypeableInt inc)
           Int $fDataInt x0)
        (everywhere
           (λ b0 $dData0 →
              mkT Int b0 ($p1Data b0 $dData0)
                $fTypeableInt inc)
           [Int] $dData xs0)
```

Since the eliminated `gmapT` is a class method, this inlining is particular to the type at which `gmapT` is applied. In this case it is over the list type, and `gmapT` inlines to a `case` expression over lists. As this `case` expression corresponds to the one in `increment`$_{\texttt{Hand}}$, we can now recognize the structure of `increment`$_{\texttt{Hand}}$ becoming manifest in the code.

The code now contains two calls to `everywhere` that are inside the `(:)` branch of the `case` expression. One is on the head of the list and is at the type `Int`. The other is on the tail of the list and is at the type `[Int]`. We can inline the first of these which results in calls to `mkT` and `gmapT` just as before. This time, however, they are over the `Int` type. Thus, not only does the `cast` in `mkT` succeed and the `mkT` reduce to `inc`, but the call to `gmapT` reduces to the identity function. After a bit of simplification, the code now looks like the following.

```
   incrementSYB :: [Int] -> [Int]
   incrementSYB = λ x →
     case x of wild
       [] → [] Int
       (:) x0 xs0 →
         (:) Int
           (inc x0)
           (everywhere
              (λ b0 $dData0 →
                 mkT Int b0 ($p1Data b0 $dData0)
                   $fTypeableInt inc)
              [Int] $dData xs0)
```

Thus far we have eliminated several runtime costs merely by inlining and some basic simplifications, and this has brought us close to our goal of transforming `increment`$_{\texttt{SYB}}$ into `increment`$_{\texttt{Hand}}$. The only generic part of the code that remains is the call to `everywhere` on the tail of the list. While it is tempting to also inline this call, this expression is the same one that `increment`$_{\texttt{SYB}}$ started with, and continuing to inline will thus lead us in a loop. Instead, we can take advantage of the fact that `increment`$_{\texttt{SYB}}$ equals this expression and replace it with a reference to `increment`$_{\texttt{SYB}}$. Once we perform that replacement, we get the following code, which is identical to that of `increment`$_{\texttt{Hand}}$.

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

4. A more principled attempt

The transformation in Section 3 is achieved by a simple combination of inlining, memoization, simplification and symbolic evaluation. In order to automate it, we must be precise about what we choose to inline, memoize, and evaluate. For a general-purpose optimization, designing such a heuristic is hard. However, because we are optimizing a particular style of code, namely SYB-style code, we can take advantage of domain-specific knowledge.

We express these transformations in terms of System F_C (Vytiniotis et al. 2012), the formal language corresponding to GHC's `Core` language. Figure 1 presents the relevant parts of the syntax of

$e, u := x$	Variables
$\mid l$	Literals
$\mid \Lambda a : \kappa.\, e \mid e\,\tau$	Type abstraction and application
$\mid \lambda x : \sigma.\, e \mid e_1\, e_2$	Term abstraction and application
$\mid K \mid \textbf{case}\, e_0\, \textbf{of}\, \overrightarrow{p_i \to e_i}$	Constructors and `case` matching
$\mid \textbf{let}\, \overrightarrow{x : \tau = e}\, \textbf{in}\, u$	Local variable binding
$\mid e \triangleright \gamma$	Casts
$\mid \lfloor \gamma \rfloor$	Coercions as expressions
$p := K\, \overrightarrow{x : \tau}$	Patterns
$\tau := a \mid \forall a : \kappa.\tau \mid \tau_1\, \tau_2 \mid \ldots$	Types
$\kappa := \star \mid \# \mid \kappa \to \kappa$	Kinds
$\gamma := \textbf{sym}\, \gamma$	Symmetry rule for coercions
$\mid \textbf{nth}\, 1\, \gamma$	Arg part of function coercion
$\mid \textbf{nth}\, 2\, \gamma$	Result part of function coercion
$\mid \gamma @ \tau$	Type application for coercions
$\mid \ldots$	

Figure 1. Syntax of System F_C (Excerpt)

System F_C, and Figure 2 presents some of the core reduction rules of System F_C. For simplicity of presentation these figures omit aspects of System F_C that are not relevant to the optimization considered in this paper. In particular, System F_C contains additional types and coercions not listed in Figure 1, as well as additional reductions and machinery for specifying the evaluation contexts for the reduction rules in Figure 2.

At a high level, the complete optimization can be summarized as follows. The details and rationale of the individual steps are explained in the remainder of this section.

Algorithm 1. *[SYB Optimization] Repeatedly loop until none of the following rules apply. On each loop choose the first rule that applies.*

1. *Replace any expression with a memoization that it matches as discussed in Section 4.2.*
2. *Simplify any expression using the rules from Figure 7 as discussed in Section 4.3.*
3. *Evaluate any primitive call using the rules from Figure 9 as discussed in Section 4.4.*
4. *(OPTIONAL) Eliminate any `case` expression over a manifest constructor as discussed in Section 4.5.1.*
5. *(OPTIONAL) Float memoization bindings if possible as discussed in Section 4.5.2.*
6. *Choose the outermost expression at which we can do either of the following as discussed in Section 4.1.*
   (a) *Memoize an expression having an undesirable type using the rules from Figure 5.*
   (b) *Eliminate an expression having an undesirable type using the rules from Figure 4.*

Note that our optimization relies on later optimizations already in GHC to further clean up the resulting code after our optimization completes. For example, it may leave behind unused memoization bindings that downstream optimizations will eliminate. In addition, steps 4 and 5 of this algorithm are optional in that they reduce the work that the optimization has to do but are not essential for eliminating expressions that have undesirable types.

BETA	$(\lambda x : \tau .\, e_1)\ e_2$	$\rightsquigarrow e_1\,[e_2/x]$
TYBETA	$(\Lambda a : \kappa .\, e)\ \tau$	$\rightsquigarrow e\,[\tau/a]$
CASEBETA	$\mathbf{case}\, K\ \vec{e_i}\ \mathbf{of}\ \ldots K\ \overrightarrow{x_i : \tau_i} \rightarrow e_j \ldots$	$\rightsquigarrow e_j\left[\overrightarrow{e_i/x_i}\right]$
PUSH	$(e_1 \triangleright \gamma)\ e_2$	$\rightsquigarrow (e_1\ (e_2 \triangleright \mathbf{sym}\ (\mathbf{nth}\,1\,\gamma))) \triangleright (\mathbf{nth}\,2\,\gamma)$
TYPUSH	$(e \triangleright \gamma)\ \tau$	$\rightsquigarrow (e\,\tau) \triangleright (\gamma @ \tau)$

Figure 2. Reductions of System F_C (Excerpt)

$e : \tau$	Expression typing
$\gamma : \tau_1 \sim \tau_2$	Coercion typing
$e \rightsquigarrow e'$	System F_C evaluation step (See Figure 2)
$e \rightarrowtail e'$	Optimization step (See Figures 4, 5 and 7)
$e \overset{\gamma}{\hookrightarrow} e'$	Symmetric cast elimination (See Figure 8)
$e \overset{.}{\rightsquigarrow} e'$	Force step (See Figures 6 and 9)
$e \overset{..}{\rightsquigarrow} e'$	Deep force step (See Figure 9)
$\mathbf{Und}\,\tau$	Undesirable type
$\mathbf{ElimUnd}\,e$	Elimination expression (See Figure 4)
$\mathbf{Memo}\,e$	Memoizable expression (See Figure 5)

Figure 3. Judgments

$$\frac{\mathbf{ElimUnd}\,e \qquad e \overset{.}{\rightsquigarrow} e'}{e \rightarrowtail e'}\ \text{ELIMUND}$$

$$\frac{e_1 : \tau_1 \rightarrow \tau_2 \qquad \mathbf{Und}\,\tau_1}{\mathbf{ElimUnd}\ (e_1\, e_2)}\ \text{ELIMUNDAPP}$$

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

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

Figure 4. Undesirably Typed Expression Elimination

$$\frac{\mathbf{ElimUnd}\,e \qquad e \overset{.}{\rightsquigarrow} e' \qquad \mathbf{Memo}\,e \qquad x \notin fv\,(e')}{e \rightarrowtail \mathbf{let}\, x : \tau = e'\, \mathbf{in}\, x}\ \text{MEMOUND}$$

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

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

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

Figure 5. Undesirably Typed Expression Memoization

With the benchmarks in Section 5 we show that this algorithm successfully optimizes typical SYB-style code to be as fast as hand-written code. Remarkably, this optimization algorithm requires no changes to the standard SYB library other than what is necessary to ensure inlining information is available for the appropriate methods, operators and traversals defined by SYB.

4.1 Elimination of expressions with undesirable types

In Section 2.4, we identified the presence of expressions with certain types as a source of performance problems in SYB-style code. However, the transformations performed in Section 3 allowed us to eliminate expressions with those types from the code for `increment`$_{\texttt{SYB}}$. One of the primary goals of our optimization then is eliminating these occurrences. In particular, objects of type `TypeRep`, as well as the `TyCon` objects used to construct them, slow down the code when they are used to compare types at runtime. In addition, the `Data` and `Typeable` dictionaries contain functions that may generate and manipulate `TypeRep` and `TyCon` objects. Finally, the default implementations of several of the methods in the `Data` class use `newtype` wrappers such as `ID` that interfere with the optimization process and should also be eliminated.

In Section 3, we were able to eliminate expressions that have these undesirable types by a combination of inlining and simplification. Moreover, the only inlining operations necessary were ones that eliminated such expressions. Thus we can design a heuristic that focuses on expressions that both have these types and are in elimination positions. Expressions in elimination positions are those that are arguments to function applications, scrutinees of `case` expressions, and the bodies of casts. If we can simplify the expression far enough to be able to apply the BETA or the CASEBETA rules in Figure 2 or expose nested casts that cancel each other out, we can eliminate those occurrences and thus remove the expressions with undesirable types from our code.

Essentially what we need to do is symbolically evaluate these expressions until they are values and then apply the appropriate reduction rules to the elimination forms. Formally this is specified by the ELIMUND rule in Figure 4. If e is an elimination form for an expression with an undesirable type and we can symbolically evaluate e to e', then the optimization simplifies e to e'. The elimination forms are specified in the ELIMUNDAPP, ELIMUNDCASE, and ELIMUNDCAST rules, and the rules for forcing a step of evaluation are specified in Figure 6. These rules use the $\mathbf{Und}\,\tau$ judgment, which holds if and only if the type τ syntactically contains an occurrence of an undesirable type. The inference rules for the $\mathbf{Und}\,\tau$ judgment are omitted as they are straightforward. In addition, we will use typing judgments for expressions, $e : \tau$, and

FORCEBETA	$(\lambda x : \tau.\, e_1)\ e_2$	$\dot{\rightsquigarrow}\ \mathbf{let}\, x : \tau = e_2\, \mathbf{in}\, e_1$	
FORCETYBETA	$(\Lambda a : \kappa.\, e)\ \tau$	$\dot{\rightsquigarrow}\ \mathbf{let}\, a : \kappa = \tau\, \mathbf{in}\, e$	
FORCECASEBETA	$\mathbf{case}\, K\, \vec{e_i}\, \mathbf{of}\ \ldots K\, \overrightarrow{x_i : \tau_i} \rightarrow e_j \ldots$	$\dot{\rightsquigarrow}\ \mathbf{let}\, \overrightarrow{x_i : \tau_i = e_i}\, \mathbf{in}\, e_j$	
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	$(\mathbf{let}\, \overrightarrow{x : \tau = e_i}\, \mathbf{in}\, e_0)\ u$	$\dot{\rightsquigarrow}\ \mathbf{let}\, \overrightarrow{x : \tau = e_i}\, \mathbf{in}\, e_0\, u$	
FORCELETFLOATSCR	$\mathbf{case}\ (\mathbf{let}\, \overrightarrow{x : \tau = u}\, \mathbf{in}\, e_0)\ \mathbf{of}\, \overrightarrow{p_i \rightarrow e_i}$	$\dot{\rightsquigarrow}\ \mathbf{let}\, \overrightarrow{x : \tau = u}\, \mathbf{in}\ (\mathbf{case}\, e_0\, \mathbf{of}\, \overrightarrow{p_i \rightarrow e_i})$	
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	$\mathbf{case}\, e_0\, \mathbf{of}\, \overrightarrow{p_i \rightarrow e_i}$	$\dot{\rightsquigarrow}\ \mathbf{case}\, e_0'\, \mathbf{of}\, \overrightarrow{p_i \rightarrow e_i}$	if $e_0\ \dot{\rightsquigarrow}\ e_0'$
FORCELETBODY	$\mathbf{let}\, \overrightarrow{x_i : \tau_i = u_i}\, \mathbf{in}\, e$	$\dot{\rightsquigarrow}\ \mathbf{let}\, \overrightarrow{x_i : \tau_i = u_i}\, \mathbf{in}\, e'$	if $e_0\ \dot{\rightsquigarrow}\ e_0'$
FORCECAST	$e \triangleright \gamma$	$\dot{\rightsquigarrow}\ e' \triangleright \gamma$	if $e\ \ \dot{\rightsquigarrow}\ e'$

Figure 6. Forcing Rules

coercions, $\gamma : \tau_1 \sim \tau_2$. These judgments respectively assert that expression e has type τ and that the coercion γ casts type τ_1 to type τ_2. The inference rules for these typing judgments are omitted as they are standard in System F_C. In these and other rules, we elide details about the environment as it is not relevant to the optimization other than to support the typing judgments.

Finally, Figure 6 gives the FORCEBETA, FORCETYBETA, FORCECASEBETA, FORCEPUSH, and FORCETYPUSH rules, which implement symbolic evaluation for the BETA, TYBETA, CASEBETA, PUSH, and TYPUSH reduction rules respectively. The FORCEBETA, FORCETYBETA, and FORCECASEBETA rules avoid code duplication by introducing `let` bindings instead of substituting. It is then up to FORCEVAR to inline forced variables at their use sites. In order to ensure that the `let` forms in the code do not interfere with the optimization process, we also introduce the rules FORCELETFLOATAPP and FORCELETFLOATSCR which float `let` bindings out of the way so that other rules can fire. The FORCEAPPFUN, FORCEAPPTYFUN, FORCESCR, FORCELETBODY, and FORCECAST rules implement structural congruences that allow the forcing process to recur down the expression. The guiding principle in all these rules is to make the smallest transformation necessary to expose an expression form that can be eliminated.

4.2 Memoization

In Section 3, we needed to recognize the repeated occurrence of `everywhere (mkT inc)` and replace it with a variable reference bound to an equivalent expression. Essentially this is a memoization of the inlining process. Without such memoization, the recursive structure of `everywhere` makes the optimization diverge.

Rather than performing a deep analysis of what inlinings and expansions should be memoized, we adopt the very simple strategy of memoizing when the expression e in ELIMUND is the application of a variable to one or more arguments. Thus we have MEMOUND in Figure 5. This rule has higher priority than ELIMUND and should be used instead of that rule whenever possible. This strategy may lead to unnecessary extra memoization bindings, but as long as those binding do not get in the way of our other optimizations, this is not a concern.

Note that we memoize inlinings only when they eliminate an expression with an undesirable type. The reason for this is that we want to memoize only code that would have triggered ELIMUND and not necessarily every intermediate expression.

When MEMOUND fires we also add e to a memoization table and if e ever occurs again, we replace it with x. We detect reoccurrences only when an expression is manifestly equal to e as we use a simple, syntactic comparison modulo alpha equivalence. For example if e is the expression `mkT f`, then we do not consider `mkT f'` to be a reoccurrence of e even if `f'` is bound to `f`. While in theory the optimization could as a result miss opportunities to take advantage of the memoization, in practice there are only a few ways that this happens in SYB-style code, and they are automatically eliminated by the other simplifications in the optimization.

4.3 Simplification

As we symbolically evaluate the code, detritus can build up in the form of dead and trivial `let` bindings and unnecessary casts. Though in some cases we can leave the elimination of these for later optimization passes in the compiler, some of these `let` bindings and casts get in the way of the core optimization rules from Figure 4 and Figure 5. In the example in Section 3, many of the intermediate simplifications were omitted in order to focus on the core aspects of the optimization, but now we formally specify these by applying the simplifications from Figure 7 to the code as we are optimizing it. These simplifications are chosen based on an empirical observation of the sort of code generated when optimizing SYB-style code and what forms need to be simplified in that process. While there are a number of other simplifications that could be used, we restrict ourselves to a minimal number of conservative simplifications that never make the code worse while still being sufficient to enable the core optimization rules.

4.3.1 Cast elimination

GHC's implementation of class dictionaries and `newtype` definitions makes use of casts. When inlining a class method or a computation that involves a `newtype`, these casts appear in the code and get in the way of the core optimization rules. For example, it often happens that the inlining of a class method results in the scrutinee

CASTREFL	$e \triangleright \gamma$	$\rightarrowtail e$ if $\gamma : \tau \sim \tau$
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	$\mathbf{let}\, x : \star = \tau\, \mathbf{in}\, e$	$\rightarrowtail e\,[\tau/x]$
SUBSTHASH	$\mathbf{let}\, x : \# = \tau\, \mathbf{in}\, e$	$\rightarrowtail e\,[\tau/x]$
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

Figure 7. Simplifications

of a `case` containing a reflexive cast wrapped around a constructor. Until we eliminate the cast, we cannot use the FORCECASEBETA rule even though the constructor involved is already manifest.

In SYB-style code, there are two sorts of such casts that arise. The first is a reflexive cast from a type to itself. These are directly eliminated by the CASTREFL rule. The second way casts can be eliminated is when symmetric casts are nested inside each other. In some cases, these symmetric casts may be separated from each other by intermediate forms as in the following example where $\gamma_1 : \tau_1 \sim \tau_2$ and $\gamma_2 : \tau_2 \sim \tau_1$.

$$(\mathbf{case}\, x\, \mathbf{of}\ \{C_1 \to e_1 \triangleright \gamma_2; C_2 \to e_2 \triangleright \gamma_2\}) \triangleright \gamma_1$$

Simplifying this expression is accomplished by the CASTSYM rule. This rule uses the $e \overset{\gamma}{\hookrightarrow} e'$ judgment in Figure 8 and reduces this expression to the following.

$$\mathbf{case}\, x\, \mathbf{of}\ \{C_1 \to e_1; C_2 \to e_2\}$$

4.3.2 Let elimination

We also eliminate `let` bindings that are either trivial, dead or bind a type as they may interfere with our ability to apply the core optimization rules. These are implemented by the remaining rules in Figure 7. Note that when doing this we are careful to not eliminate bindings introduced by memoization. In particular, due to the way that GHC implements class dictionaries, it is quite common for a memoized call to expand to another memoized call in a way that results in the memoized binding for the original call becoming trivial. We must avoid eliminating these as the memoization process may add new references to such bindings.

4.4 Primitives

Recall that the `cast` function is implemented by testing the equality of two `TypeRep` objects returned by calls to `typeOf`. This `typeOf` operator is implemented in terms of `fingerprintFingerprints`, which computes unique hashes for `TypeRep` objects. Furthermore, equality over these objects is implemented in terms of the `eqWord#` primitive. As we are attempting to eliminate the dynamic dispatches implemented by `cast`, it is important that we eliminate calls to these primitives. In order to do so, our optimization fully evaluates the arguments to these functions when attempting to force an expression. Once those arguments are fully evaluated, the calls themselves are statically evaluated. The rules that implement this are specified in Figure 9. These rules effectively implement constant folding for these operators.

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

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

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

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

Figure 8. Cast Symmetry Rules

4.5 Optional optimizations

While not essential to the core optimization and the elimination of expressions with undesirable types, there are certain transformations that help keep the generated code compact and reduce the amount of work to be done by the optimization.

4.5.1 Case reduction

SYB-style traversals are based on the idea of dispatching to different code depending on the current type being traversed. At its core, this is what the `mkT` function is for. When optimizing SYB-style code, this often results in intermediate residual code with a structure similar to the following.

```
case typeOf t1 == typeOf t2 of
  True  -> ...
  False -> ...
```

The equality operator in this code is over the undesirable type `TypeRep`, so the optimization will reduce it to either `True` or `False`. After that, the scrutinee no longer contains an expression with an undesirable type, so the core optimization does not then simplify the `case` expression even though it has a known constructor in its scrutinee. In most cases this is not a problem as the code to be optimized under each branch of the `case` expression tends to be small and we can simply rely on downstream optimizations to simplify the `case` expression. However, when these branches are large, they can represent a significant amount of extra work to be done by the optimization. It would be better to detect the dead branch and skip the extra work in that branch. To do this we apply the rewrite in FORCECASEBETA whenever possible. This rewrite never makes the code worse or worsens the optimization result. Note that our use of this rewrite differs from the usual use of the rules in Figure 6 since we apply it at any position in the expression regardless of whether it eliminates an expression with an undesirable type.

4.5.2 Memoization floating

Duplicate memoizations of the same expression may arise if the first memoization is not in scope at the other occurrences of the same expression. For example, when traversing an abstract syntax tree, memoizations of the traversal at the identifier type may occur inside both the part of the code for λ-expressions and the part of the code for `let` expressions. If neither of these is within the scope of the other, the memoization rule will result in creating fresh memoizations of the traversal on identifiers for each expression

PRIMFF	`fingerprintFingerprints` e	$\overset{\cdot}{\rightsquigarrow}$ $[\![$`fingerprintFingerprints` $e]\!]$	if e is a value
PRIMFFARG	`fingerprintFingerprints` e	$\overset{\cdot}{\rightsquigarrow}$ `fingerprintFingerprints` e'	if $e \overset{\cdot\cdot}{\rightsquigarrow} e'$
PRIMEQWORD	`eqWord#` $e_1\ e_2$	$\overset{\cdot}{\rightsquigarrow}$ $[\![$`eqWord#` $e_1\ e_2]\!]$	if e_1 and e_2 are values
PRIMEQWORDARG1	`eqWord#` $e_1\ e_2$	$\overset{\cdot}{\rightsquigarrow}$ `eqWord#` $e_1'\ e_2$	if $e_1 \overset{\cdot\cdot}{\rightsquigarrow} e_1'$
PRIMEQWORDARG2	`eqWord#` $e_1\ e_2$	$\overset{\cdot}{\rightsquigarrow}$ `eqWord#` $e_1\ e_2'$	if $e_2 \overset{\cdot\cdot}{\rightsquigarrow} e_2'$
FORCEDEEP	e	$\overset{\cdot\cdot}{\rightsquigarrow} e'$	if $e \overset{\cdot}{\rightsquigarrow} e'$
FORCEDEEPARG	$e_1\ e_2$	$\overset{\cdot\cdot}{\rightsquigarrow} e_1\ e_2'$	if $e_2 \overset{\cdot\cdot}{\rightsquigarrow} e_2'$

Figure 9. Rules for Primitives

form even though the code for these memoizations are identical to each other.

As a consequence of this, it is relatively easy to get code that is exponentially large in the size of the types being traversed because the inlining process may not terminate until every path down the expanded expression contains a memoization for every type being traversed. Even in cases when the code does not blow up to be exponentially large, these duplicated memoizations represent extra work for the optimizer and inflate the size of the resulting code.

To avoid this size explosion, we `let`-float memoized bindings as far outward as possible. By floating the memoized bindings outwards, we maximize their scope and thus avoid creating duplicate memoizations due to already created memoizations being out of scope. For example, once the memoization created for the identifier in a λ-expression floats outwards, the traversal for the identifier in a `let` expression can use the existing memoization instead of creating a new one. We also consolidate memoization bindings into a common recursive `let` binding when possible as, while they may not initially refer to each other, the process of replacing expressions with their memoized bindings may make them refer to each other at some later point.

5. Implementation

We implemented the custom optimization pass described in Section 4 using HERMIT, a recently developed GHC plugin for applying transformations to `Core` (Farmer et al. 2012; Sculthorpe et al. 2013a). HERMIT was used interactively to gain an intuition about the transformations necessary, and was then extended with new primitive transformations implementing the rules given in Section 4. The overall optimization algorithm is expressed as a HERMIT script that performs Algorithm 1 and then uses HERMIT's `bash` command to simplify the code and eliminate things like dead `let` bindings that are left behind by Algorithm 1.

HERMIT provides several facilities to ease the implementation of `Core`-to-`Core` transformations such as our optimization. This includes KURE, a strategic rewriting library allowing transformations to be expressed in a high-level, declarative style (Gill 2009; Farmer et al. 2012; Sculthorpe et al. 2013b), a versioning kernel which manages the application of rewrites, congruence combinators for `Core` which automatically update the rewriting context, error reporting facilities, and a large set of existing primitive rewrites and queries. Not including primitive transformations already available in HERMIT, the entire optimization was implemented in approximately 400 lines of Haskell and did not require any modifications to GHC itself.

5.1 Benchmarks

We applied the optimization to a selection of benchmarks taken from the Haskell generic-programming literature. The resulting programs were benchmarked using a version of the framework from Magalhães et al. (2010) that was adapted to support compilation with HERMIT. The benchmarks were as follows.

RmWeights Taken from `GPBench` (Rodriguez et al. 2008), the `RmWeights` benchmark traverses a weighted binary tree while removing the weight annotations. It is implemented in SYB using the `everywhere` and `mkT` combinators.

SelectInt Also from `GPBench`, `SelectInt` traverses a weighted binary tree while collecting all the `Int`s into a single list. It is naively implemented in SYB using the `everything` and `mkQ` combinators, but as we discuss in Section 5.2, it had to be modified to ensure a fair comparison.

Map Found in Magalhães et al. (2010), `Map` performs a mapping over a structure. It is implemented in SYB using `everywhere` and `mkT`. This traversal is performed on three data types. The first is a binary tree of integers. The second is a logic formula. The third is an AST type from the `haskell-src` module involving over 30 types and 100 different constructors. For the binary tree, all integers are incremented. For the other two types, all characters are replaced with the character '`y`'.

RenumberInt Taken from Adams and DuBuisson (2012), the `RenumberInt` benchmark replaces each integer in a structure with a new, unique integer that is drawn from a state monad. This traversal is also performed on both binary tree and logic formula data types. It is implemented in SYB using `everywhereM` and `mkM`.

5.2 Benchmark setup

Each benchmark was implemented both non-generically (Hand) and using SYB combinators (SYB). The SYB implementation was also benchmarked with our optimization (SYBOPT). The benchmarking framework used in Magalhães et al. (2010) was used to run each program 10 times and take the average running time. We compiled the benchmarks with GHC HEAD[1] using the `-O2` compiler option and ran them with the `-K1g` RTS option on a 2.3 GHz, 64-bit Intel i5 with 4 GB of RAM running Darwin 11.4.2.

The implementation of `SelectInt` in `GPBench` uses two different algorithms for the Hand and SYB implementations. The Hand implementation uses a linear-time, accumulating-style traversal, while the SYB implementation uses a quadratic-time, non-accumulating traversal. To ensure a fair comparison, we modified the SYB implementation to use an accumulating traversal.

[1] A space leak in GHC was discovered during the course of this work. A patch for this that makes it possible to run the HERMIT transformations involved in this optimization was merged into GHC HEAD on June 4, 2013.

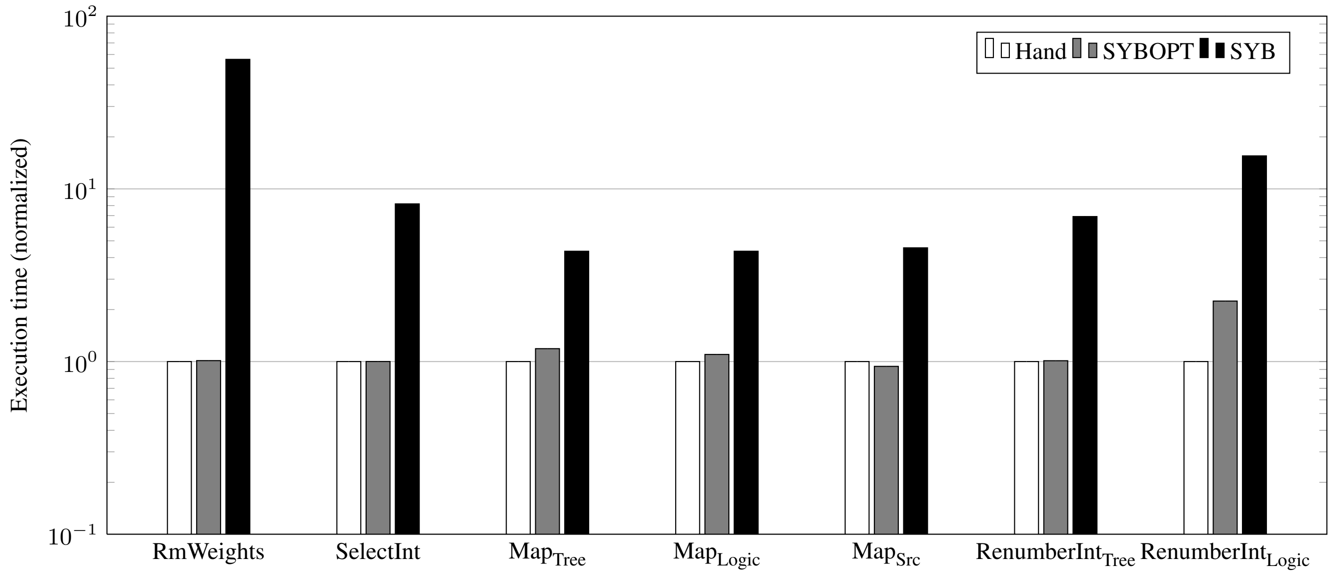

Figure 10. Benchmarks Results

5.3 Performance results

Figure 10 summarizes the resulting execution times of the benchmarks. The results are normalized relative to the Hand version and are displayed on a logarithmic scale in order to accommodate the large differences between execution times. These benchmarks confirm previous results about the poor performance of SYB as it performed on average an order of magnitude slower than the handwritten code.

For all of the benchmarks except `RenumberInt`$_{Logic}$, the optimization completely eliminates the runtime costs associated with SYB. When initially running these benchmarks, the SYBOPT versions of `Map`$_{Tree}$ and `Map`$_{Logic}$ actually ran *faster* than the Hand versions by about 20%. Analysis of the resulting `Core` revealed that, as a side effect of our optimization, the traversal was being specialized to the particular function being mapped over the structure. The Hand version did not do this. Rewriting the Hand version by applying a static-argument transformation (Santos 1995) improved its performance to match that of the SYBOPT version.

For `RenumberInt`$_{Logic}$, the SYBOPT version is 2.2 times slower than the Hand version. An examination of the generated `Core` leads us to believe that this is due to GHC not optimizing the monadic operations in the SYBOPT version as well as it does in the Hand version.

For all of the benchmarks, a manual inspection of the generated `Core` confirms that the optimization does indeed eliminate all runtime type checks and dictionary dispatches in the SYB-style code and that the resulting code is equivalent to the handwritten code.

6. Limitations and Future Work

While the algorithm described in Section 4 is effective for most instances of SYB-style code, it does have limitations and areas that future work can improve. Many of these problems will be familiar to the partial-evaluation community. As these are active research topics in their own right, we do not attempt a general solution to them but where possible note how they can be mitigated for our particular optimization. As it is domain-specific, this optimization may not be appropriate for all code, and the compiler may require assistance from the programmer in the form of pragmas or annotations to determine when to use or not use this optimization.

6.1 Missing inlining information

The first and most obvious limitation is that this optimization relies heavily on inlining and thus depends on having the appropriate inlining information available. If that information is not available, then the optimization may fail to complete its task of eliminating expressions with undesirable types. Fortunately, this is an easily detected situation, and the optimization can abort while leaving the original code intact and issuing a warning so the user can make appropriate adjustments to expose the necessary inlining information.

Missing inlining information can be caused by using functions from imported modules for which GHC has not recorded inlining information. It may also be caused by running the optimization over code in which the types over which `Data` or `Typeable` are quantified are underspecified. For example, consider the following code that someone might write as a helper function.

```
mapSYB :: (Data a) => (a -> a) -> [a] -> [a]
mapSYB f = everywhere (mkT f)
```

Since this function is polymorphic in `a`, there is no concrete dictionary available for the class constraint `Data a`, and we cannot fully optimize this function.

There are, however, two important points to consider about this limitation. First, as it is obviously impossible to specialize a generic traversal when we do not yet know the type at which to specialize, this limitation is inherent in the optimization task and not merely a failure of the optimization algorithm. For example, if `a` is instantiated with `[Char]`, then `f` must be applied not only to the elements of the list passed to `mapSYB` but also to the sub-lists of those elements. Until we know `a`, it is impossible to know how to traverse those elements.

Second and more importantly, this limitation is not a problem in practice. It simply means that the optimization must be deferred to uses of the function that specify types at which to specialize. For example, instead of optimizing `mapSYB`, we optimize uses of `mapSYB` such as the following.

```
incrementSYB/Int :: [Int] -> [Int]
incrementSYB/Int = mapSYB inc
```

Because this definition completely determines the type of a in $\texttt{map}_{\texttt{SYB}}$ and thus calls $\texttt{map}_{\texttt{SYB}}$ with a concrete Typeable dictionary for a, the optimization will successfully complete on $\texttt{increment}_{\texttt{SYB/Int}}$ even though it would fail on $\texttt{map}_{\texttt{SYB}}$.

Finally, note that specialized versions of $\texttt{map}_{\texttt{SYB}}$ can be explicitly generated by specifying their types as in the following.

```
mapSYB/Int :: (Int -> Int) -> [Int] -> [Int]
mapSYB/Int = mapSYB
```

6.2 Essential occurrences of undesirable types

Since the primary design heuristic behind this optimization is the elimination of expressions that have undesirable types, it will fail if there are expressions that should not be eliminated. An obvious example is when the type being traversed itself contains undesirable types such as TypeRep or TyCon, but less obvious examples of this include types like the following.

```
data Spine b
  = Unit b
  | ∀a. (Data a) =>
      App (Spine (a -> b)) a
```

Here the existential[2] type a is qualified by the Data class and thus the App constructor contains a dictionary for the Data class.

Along similar lines, it may be possible for a particular traversal to contain essential uses of undesirable types. For example, SYB allows code to arbitrarily synthesize TypeRep and TyCon objects. This may result in occurrences of undesirable types that are essential to the traversal and either should not or cannot be eliminated. Note that though such a traversal is possible, it is exceedingly rare in SYB-style code. None of the standard traversals exhibit such a structure.

This limitation may be mitigated by annotating the code with information about which occurrences of undesirable types are genuinely undesirable and which are not. Then as the optimization transforms the code, we can keep careful account of each occurrence and whether it is genuinely undesirable.

6.3 Polymorphic recursion in types

As with other forms of partial evaluation, polymorphic recursion is a concern with this optimization. Most types in Haskell programs are regular, but non-regular, polymorphically recursive types do occasionally occur. Consider, for example, the following polymorphically recursive, non-regular type.

```
data T a
  = Base a
  | Double (T (a, a))
```

If we attempt to traverse over the type T Int, then the traversal will initially be memoized at T Int. Since at this type the argument to the Double constructor is of type T (Int, Int), the traversal will also have to be memoized at type T (Int, Int). In turn, at that type the argument to the Double constructor has type T ((Int, Int), (Int, Int)), and so on. Naively running the optimization on this type would thus continue forever as the memoization process depends on the assumption that there are a finite number of types to be traversed, but the T Int type effectively contains an infinite number of types.

In order to successfully handle this, we would need to account for the fact that in many cases a non-generic traversal over a polymorphic type must be structured differently from a generic traversal. In these cases it is impossible to generate non-generic code that naively mirrors the structure of the generic code. For example, consider a traversal that increments all values of type Int inside an object of type T Int. The generic code for this is the following.

```
incrementT :: T Int -> T Int
incrementT x = everywhere (mkT inc) x
```

Now consider how one would write this with non-generic code. The recursion over the elements of T cannot have type T Int -> T Int since the Double constructor changes the type argument of T. On the other hand the recursion cannot have type ∀a. T a -> T a since being polymorphic in a prevents the function from manipulating the Int that occur in a. Instead, a more sophisticated implementation such as the following is necessary.

```
incrementT :: T Int -> T Int
incrementT x = go inc x where
  go :: (a -> a) -> T a -> T a
  go f (Base x) = f x
  go f (Double t) = Double (go (f' f) t)
  f' :: (a -> a) -> (a, a) -> (a, a)
  f' f (x1, x2) = (f x1, f x2)
```

Since the optimization presented in this paper preserves the structure of the generic traversal and $\texttt{increment}_{\texttt{T}}$ does not follow that structure, it is unsurprising that our optimization fails on such a traversal. However, note that the f argument to go serves essentially the same role as the Data dictionary in the generic traversal in that it provides the necessary information for implementing the parts of the traversal that operate over the type a. Thus an interesting direction for future work would be deriving such a non-generic implementation from the generic traversal by appropriately specializing and simplifying the Data dictionary.

6.4 Polymorphic recursion in terms

In addition to types being polymorphically recursive, the traversal itself may be polymorphically recursive in an argument whose type contains undesirable types. Traversals like this are rare in SYB-style code, but one could imagine an example like the following.

```
poly :: (∀b. Data b => b -> b)
     -> (∀a. Data a => a -> a)
poly f x = f (gmapT (poly (f ‘extT‘ g)) x)
  where g = ...
```

Note how the f argument to the traversal is extended each time through the traversal. As a result, the previously memoized instances of poly cannot be used and the optimization algorithm will never be able to completely eliminate all expressions with undesirable types.

Of course, this is a concern only because the type of f contains an undesirable type. Parameters such as x that do not have a type containing an undesirable type can freely vary from call to call as the memoization does not care about them.

6.5 Selective traversal

An instance where the optimization does not fail but the results could be improved is when parts of the generic traversal expand to trivial traversals that do no useful work. For example, a traversal that modifies only integers can safely skip over any strings that it finds and avoid processing the individual characters in the string. Adams and DuBuisson (2012) call this selective traversal and document the significant performance improvements this can achieve. SYB does not do selective traversal unless it is explicitly told what expressions to skip. In the code produced by our optimization, these skippable parts of the traversal are manifest as functions that do a

[2] GHC uses the ∀ keyword for both existential and universal types. The distinction between the two is where the keyword is placed.

trivial deconstruction and reconstruction. For example, in a traversal that effects only integers, we might find code for traversing strings similar to the following.

```
memo_Char   c        = c
memo_String []       = []
memo_String (c : cs) = memo_Char c : memo_String cs
```

Here $\texttt{memo}_{\texttt{String}}$ is equivalent to the identity function and can thus be more efficiently implemented by not doing the traversal and simply returning its argument. Depending on the structure of the data being traversed, this can lead to significant speedups.

Similar situations arise for queries and monadic traversals. For queries, some parts of the traversal may produce trivial query results, and for monadic traversals, some parts of the traversal may be equivalent to simply applying `return` to the tree being traversed.

Identifying and optimizing these trivial functions is fairly easy and can be done by a post-processing pass after our optimization. We plan to add this in future versions of our implementation.

7. Related work

Generic-programming systems in Haskell are often slow relative to handwritten code. There has been a significant amount of work on designing more efficient generic-programming systems (Mitchell and Runciman 2007; Brown and Sampson 2009; Chakravarty et al. 2009; Augustsson 2011; Adams and DuBuisson 2012), but there is little work on optimizing a pre-existing generic-programming system as we do here. Magalhães (2013) shows how to optimize the `generic-deriving` system by using standard compiler optimizations, but notes that his techniques are not sufficient to optimize SYB-style code. Alimarine and Smetsers (2004) have developed a similar optimization system for generics in the Clean language.

In a broad sense, our optimization is a form of partial evaluation (Jones et al. 1993) with a binding-time analysis that uses type information to determine whether code should be statically computed at compile time or dynamically evaluated at runtime. However, because we use domain specific knowledge, our algorithm can be simpler and more direct than traditional partial evaluation. This idea is also related to the partial evaluation of class dictionaries (Jones 1995), and can be seen as a form of call-pattern specialization (Peyton Jones 2007). However, our optimization specializes and memoizes over any expression with an undesirable type whereas Jones (1995) specializes over only class dictionaries, and Peyton Jones (2007) specializes over only manifest constructors.

Finally, our optimization can be seen as a limited form of supercompilation (Turchin 1979). Like Bolingbroke and Peyton Jones (2010), we implement a memoization scheme to ensure terms are optimized only once, but we can more easily direct the optimization as we restrict ourselves to optimizing SYB-style code. In theory, we face the same problem of code explosion that supercompilers do (Jonsson and Nordlander 2011), but as we operate in the more limited setting of SYB-style code, this problem is easier to handle.

8. Conclusion

SYB is widely used in the Haskell community. Its poor performance, however, can be a serious drawback in practical systems. Nevertheless, by using domain specific knowledge about SYB-style code, we can design an optimization that transforms this code to be as fast as equivalent handwritten, non-generic code.

The essential task of this optimization is the elimination of certain types by a compile-time symbolic evaluation of the appropriate parts of the code. We have implemented this optimization in the HERMIT plugin for GHC. The interactive manipulation that HERMIT supports made it easy to rapidly prototype such an optimization and trace how it transforms the code. This interactive approach was instrumental in empirically discovering the appropriate optimization steps for optimizing SYB-style code. For example, a number of auxiliary code simplifications had to be introduced in order to make it possible for the core rules to run. In the future, we hope to explore how to integrate this optimization directly into GHC, and how to make it applicable to other domains where expressions of certain types need to be eliminated.

Benchmarks show that this optimization significantly improves the performance of several typical SYB-style traversals to closely match that of handwritten, non-generic code. In so doing, this optimization changes SYB from being one of the slowest generic-programming systems in the Haskell community to being one of the fastest.

Acknowledgments

The work presented in this paper was supported in part by the National Science Foundation (NSF) grants CCF-1218605 and 1117569, the Rockwell Collins contract 4504813093, and the Engineering and Physical Sciences Research Council (EPSRC) grant EP/J010995/1.

References

Michael D. Adams and Thomas M. DuBuisson. Template your boilerplate: using Template Haskell for efficient generic programming. In *Proceedings of the 2012 symposium on Haskell symposium*, Haskell '12, pages 13–24, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1574-6. doi: 10.1145/2364506.2364509.

Artem Alimarine and Sjaak Smetsers. Optimizing generic functions. In Dexter Kozen, editor, *Mathematics of Program Construction*, volume 3125 of *Lecture Notes in Computer Science*, pages 16–31. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-22380-1. doi: 10.1007/978-3-540-27764-4_3.

Lennart Augustsson. Geniplate version 0.6.0.0, November 2011. URL `http://hackage.haskell.org/package/geniplate/`.

Maximilian Bolingbroke and Simon Peyton Jones. Supercompilation by evaluation. In *Proceedings of the third ACM Haskell symposium on Haskell*, Haskell '10, pages 135–146, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0252-4. doi: 10.1145/1863523.1863540.

Neil C. C. Brown and Adam T. Sampson. Alloy: fast generic transformations for Haskell. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, Haskell '09, pages 105–116, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-508-6. 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.

Andrew Farmer, Andy Gill, Ed Komp, and Neil Sculthorpe. The HERMIT in the machine: A plugin for the interactive transformation of GHC core language programs. In *2012 ACM SIGPLAN Haskell Symposium*, pages 1–12, New York, 2012. ACM.

GHC Team. *The Glorious Glasgow Haskell Compilation System User's Guide, Version 7.6.2*, 2013. URL `http://www.haskell.org/ghc`.

Andy Gill. A Haskell hosted DSL for writing transformation systems. In Walid Mohamed Taha, editor, *Domain-Specific Languages*, volume 5658 of *Lecture Notes in Computer Science*, pages 285–309. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-03033-8. doi: 10.1007/978-3-642-03034-5_14.

Industrial Haskell Group. Hackage: Total downloads, 2013. URL `http://hackage.haskell.org/packages/top`. Accessed on October 8, 2013.

Mark P. Jones. Dictionary-free overloading by partial evaluation. *LISP and Symbolic Computation*, 8(3):229–248, September 1995. ISSN 0892-4635 (Print) 1573-0557 (Online). doi: 10.1007/BF01019005.

Neil D. Jones, Casrten K. Gomard, and Peter Sestof. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International Series in Computer Science. Prentice Hall, 1993. ISBN 978-0-13-020249-9.

Peter A. Jonsson and Johan Nordlander. Taming code explosion in supercompilation. In *Proceedings of the 20th ACM SIGPLAN workshop on Partial evaluation and program manipulation*, PEPM '11, pages 33–42, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0485-6. doi: 10.1145/1929501.1929507.

Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, TLDI '03, pages 26–37, New York, NY, USA, 2003. ACM. ISBN 1-58113-649-8. doi: 10.1145/604174.604179.

Ralf Lämmel and Simon Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *Proceedings of the ninth ACM SIGPLAN international conference on Functional programming*, ICFP '04, pages 244–255, New York, NY, USA, 2004. ACM. ISBN 1-58113-905-5. doi: 10.1145/1016850.1016883.

José Pedro Magalhães. Optimisation of generic programs through inlining. In *Implementation and Application of Functional Languages*, 2013.

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*, PEPM '10, pages 33–42, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-727-1. doi: 10.1145/1706356.1706366.

Neil Mitchell and Colin Runciman. Uniform boilerplate and list processing. In *Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, Haskell '07, pages 49–60, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-674-5. doi: 10.1145/1291201.1291208.

Simon Peyton Jones. Call-pattern specialisation for Haskell programs. In *Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, ICFP '07, pages 327–337, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-815-2. doi: 10.1145/1291151.1291200.

Simon Peyton Jones and Simon Marlow. Secrets of the Glasgow Haskell Compiler inliner. *Journal of Functional Programming*, 12(4–5):393–434, July 2002. ISSN 1469-7653. doi: 10.1017/S0956796802004331.

Alexey Rodriguez, Johan Jeuring, Patrik Jansson, Alex Gerdes, Oleg Kiselyov, and Bruno C. d. S. Oliveira. Comparing libraries for generic programming in Haskell. Technical Report UU-CS-2008-010, Utrecht University, 2008.

Alexey Rodriguez Yakushev. *Towards Getting Generic Programming Ready for Prime Time*. PhD thesis, Utrecht University, 2009.

André Santos. *Compilation by Transformation in Non-Strict Functional Languages*. PhD thesis, University of Glasgow, 1995.

Neil Sculthorpe, Andrew Farmer, and Andy Gill. The HERMIT in the tree: Mechanizing program transformations in the GHC core language. In *Implementation and Application of Functional Languages*, 2013a.

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, 2013b.

V[alentin] F[yodorovich] Turchin. A supercompiler system based on the language REFAL. *ACM SIGPLAN Notices*, 14(2):46–54, February 1979. ISSN 0362-1340. doi: 10.1145/954063.954069.

Dimitrios Vytiniotis, Simon Peyton Jones, and José Pedro Magalhães. Equality proofs and deferred type errors: a compiler pearl. In *Proceedings of the 17th ACM SIGPLAN international conference on Functional programming*, ICFP '12, pages 341–352, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1054-3. doi: 10.1145/2364527.2364554.