Optimizing SYB Is Easy!

Michael D. Adams
Department of Computer Science,
Portland State University
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 about 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. The optimization takes advantage of domain-specific knowledge about the structure of SYB and in so doing can optimize SYB-style traversals to be as fast as handwritten, non-generic code.

This paper presents both the formal structure of the optimization and the results of benchmarking the optimized SYB code against both unoptimized SYB code and handwritten, non-generic code. In these benchmarks, the optimized SYB code matches the performance of handwritten code even when the unoptimized SYB code is an order of magnitude or more slower.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Functional Programming

General Terms Algorithms, Languages

Keywords benchmark, datatype-generic programming, Haskell, optimization, SYB

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. In the Hackage software archive, it is among the 1.5% most depended upon packages [Snoyman 2013]. It has strong support from the GHC compiler and is easy to use.

A disadvantage of SYB, however, is its performance. SYB's poor runtime performance relative to other generic programming systems is well documented [Rodriguez Yakushev 2009; Brown and Sampson 2009; Chakravarty et al. 2009; Magalhães et al. 2010; Adams and DuBuisson 2012; Sculthorpe et al. 2013b], and our own benchmarks show it to be an order of magnitude slower than handwritten, non-generic code.

Other generic-programming systems deliver better performance than SYB, but SYB remains the most popular generic-programming library in the Haskell community. While SYB allows the easy and concise expression of traversals over large types that would otherwise require large amounts of handwritten code, its slow runtime performance represents a serious drawback to its use in many practical systems.

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 exhibits a structure that we can take advantage of in our optimizations. This paper presents a domain-specific optimization that uses knowledge of this structure to transform SYB code such that it is as fast as handwritten code. This allows the programmer to write high-level, generic code without the performance costs usually associated with SYB.

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 [Bolingbroke and Peyton Jones 2010] or 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. Not only does this paper present an algorithm for optimizing SYB code, but it also serves as a case study in domain specific optimization and the use of interactive optimization systems like HERMIT [Farmer et al. 2012; Sculthorpe et al. 2013a] to develop such optimizations.

The remainder of this paper is organized as follows. We start with an introduction to SYB in Section 2. In Section 3 we show a step-by-step "manual" optimization of an SYB program, followed by the full description of our automatic optimization system in Section 4. In Section 5 we discuss an implementation of the optimization for the Glasgow Haskell Compiler (GHC) [GHC Team 2013], and benchmark the results to validate its effectiveness. This is followed by a discussion of the limitations of our system in Section 6. Finally, we discuss future work in Section 7, review related work in Section 8, and conclude in Section 9.

2. Overview of SYB

In order to understand why SYB is slow, we must first understand how it works. SYB provides the user with a number of powerful predefined generic functions. At the core of these are the `gfoldl` method of the `Data` class and the `cast` operator implemented in

terms of the `typeOf` method of the `Typeable` class. Class instances for both the `Data` and `Typeable` classes can be automatically derived by the compiler. The `gfoldl` function deconstructs terms in order to gain access to their children, while the `cast` and `typeOf` operators enable dynamic dispatch based on the type of a term. These functions can be used to define higher-level operators that perform various generic traversals and transformations. An example of such a high-level generic function is `everywhere`, which traverses a structure without changing its type, and is implemented as follows.

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

This function applies `f` to all the subterms of `x` in a bottom-up fashion. It first uses `gmapT` to apply `everywhere f` to every subterm of `x`, and afterwards it applies `f` to the result. The function `gmapT` is part of the `Data` class, and has a default implementation in terms of `gfoldl`, which in turn is defined automatically by GHC's deriving mechanism. The `gmapT` function applies a transformation to all the immediate subterms of a given term, and we discuss its implementation in Section 2.2. It does not itself recurse past the first layer of children, but by calling it with `everywhere f` as an argument, the `everywhere` recurses to all the descendants of `x`.

We can use `everywhere` to define a traversal that increments all the integers in a list of integers as with the following code.

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

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

Since this transformation uses both `mkT` directly and `gmapT` indirectly through `everywhere`, we now turn to how these functions 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 setting is safe because of the check that the types are indeed the same.

2.2 Mapping subterms

To apply the transformation to all subterms, `everywhere` uses `gmapT`. 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. It is implemented 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 x_1...x_n) = z C `k` x_1 ... `k` x_n
```

While extremely general, `gfoldl` is not easy to use directly. However, generic functions that are easier to use can built in terms of it. An example is `gmapT`, which has a default implementation defined 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 of `ID`, `gmapT` operates by using `k` to rebuild the constructor application after applying `f` to each constructor argument. For any constructor `C` with n arguments, `gmapT` thus obeys the following equality.

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

2.3 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], who developed a new generic-programming library because SYB was too slow, found SYB to be 4 to 23 times slower than their own approach. Magalhães et al. [2010] report SYB to perform between 3 and 20 times slower than handwritten code. Adams and DuBuisson [2012], who developed an optimized variant of SYB using Template Haskell, report SYB to perform between 10 and nearly 100 times slower than handwritten code in a benchmark of eight generic functions. 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.

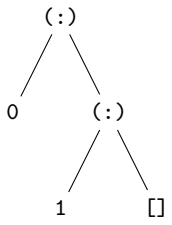

Figure 1. Tree Structure of `[0,1]`

After analyzing how SYB works, these results should not be surprising. Consider for example, the runtime behavior of the $\texttt{increment}_{\texttt{SYB}}$ function. When applied to a value of type `[Int]` such as `[0,1]`, it recurses down the structure applying `mkT inc` to every subterm. This structure is shown in Figure 1. In this case, there are five subterms. Three of them are the lists `[0,1]`, `[1]` and `[]`. The remaining two are the values `0` and `1`. For each subterm, `mkT` attempts to cast `inc` to have a type that is applicable to the 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.

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 checker 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 2.3.

```
increment_Hand :: [Int] -> [Int]
increment_Hand [] = []
increment_Hand (x : xs) =
    inc x : increment_Hand 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 b $dData →
       mkT Int b ($p1Data b $dData) $fTypeableInt inc)
    [Int]
    $dData
    x
```

Explicit type arguments are denoted here in green. Also, we will elide any type coercions as they can make the code hard to read.

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 we do not know the type of all of the arguments to which `mkT` is applied. Instead, in order to get `mkT` to a fully applied position, we inline `everywhere`, the function to which `mkT` 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 b0 $dData1 →
          everywhere
            (λ b $dData →
               mkT Int b
                 ($p1Data b $dData) $fTypeableInt inc)
            b0
            $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 b0 $dData1 →
               everywhere
                 (λ b $dData →
                    mkT Int b
                      ($p1Data b $dData)
                      $fTypeableInt
                      inc)
                 b0
                 $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 a

bit of simplification, 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 b0 $dData1 →
       everywhere
         (λ b $dData →
            mkT Int b
              ($p1Data b $dData) $fTypeableInt inc)
         b0
         $dData1)
    x
```

Here again we choose not to inline `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 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)
```

Since `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 $\texttt{increment}_{\texttt{Hand}}$, we can now recognize the structure of $\texttt{increment}_{\texttt{Hand}}$ 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 resulting 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 `gmapT` inlines 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 b $dData →
              mkT Int b
                ($p1Data b $dData) $fTypeableInt inc)
           [Int]
           $dData
           xs0)
```

$e, u := x$	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
$\mid K \mid \mathbf{case}\, e_0\, \mathbf{of}\, \overrightarrow{p_i \to e_i}$	Constructors and `case` matching
$\mid \mathbf{let}\, \overrightarrow{x : \tau = e}\, \mathbf{in}\, u$	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
$\gamma := \mathbf{sym}\, \gamma$	Symmetry rule for coercions
$\mid \mathbf{nth}\, 1\, \gamma$	Select argument part of a function coercion
$\mid \mathbf{nth}\, 2\, \gamma$	Select result part of a function coercion
$\mid \gamma @ \tau$	Type application for coercions
$\mid \ldots$	

Figure 2. Syntax of System F_C (Excerpt)

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 $\texttt{increment}_{\texttt{SYB}}$ into $\texttt{increment}_{\texttt{Hand}}$. 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 $\texttt{increment}_{\texttt{SYB}}$ started with, and continuing to inline will thus lead us in a loop. Instead, we can take advantage of the fact that $\texttt{increment}_{\texttt{SYB}}$ equals this expression and replace it with a reference to $\texttt{increment}_{\texttt{SYB}}$. Once we perform that replacement, we get the following code, which is identical to that of $\texttt{increment}_{\texttt{Hand}}$.

```
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 and Peyton Jones 2011; Vytiniotis et al. 2012], the formal language corresponding to GHC's `Core` language. Figure 2 presents the relevant parts of the syntax of System F_C, and Figure 3 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 2, as well as additional reductions and machin-

BETA	$(\lambda x : \tau.\, e_1)\ e_2$	$\rightsquigarrow \mathbf{let}\, x : \sigma = e_1\, \mathbf{in}\, e_2$
TYBETA	$(\Lambda a : \kappa.\, e)\ \tau$	$\rightsquigarrow e\,[\tau/a]$
CASEBETA	$\mathbf{case}\, K\, \vec{e_i}\, \mathbf{of}\ \ldots\, K\, \overrightarrow{x_i : \tau_i} \to e_j \ldots$	$\rightsquigarrow \mathbf{let}\, \overrightarrow{x_i : \tau_i = e_i}\, \mathbf{in}\, e_j$
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 3. Reductions of System F_C (Excerpt)

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

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

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

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

Figure 4. Undesirable Type Elimination

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

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

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

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

Figure 5. Undesirable Type Memoization

ery for specifying the evaluation contexts for the reduction rules in Figure 3.

4.1 Elimination of undesirable types

In Section 2.3, we identified the runtime manipulation of certain types as a source of performance problems in SYB-style code. However, the transformations performed in Section 3 allowed us to eliminate those types from the code for `incrementSYB`. One of the primary goals of our optimization then is eliminating occurrences of these types. We consider it undesirable to have them in our code as they lead to slow runtime performance. 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 these undesirable types by a combination of inlining and simplification. Moreover, the only inlining operations necessary were ones that eliminated an occurrence of these undesirable types. 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 or expose nested casts that cancel each other out, we can eliminate those occurrences and thus remove the 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 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 τ contains an occurrence of an undesirable type. The inference rules for this judgment are omitted as they are straightforward. In addition, we will use typing judgments for expressions, $\vdash e : \tau$, and coercions, $\vdash^{\text{co}} \gamma : \tau_1 \sim \tau_2$. 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. Additionally, FORCEVAR inlines 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 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. We can view this as a sort of memoization of the inlining process. Without such memoization, the recursive structure of `everywhere` would make the transformation diverge.

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	$\mathbf{case}\, K\, \vec{e_i}\, \mathbf{of}\ \ldots K\, \overrightarrow{x_i : \tau_i} \to e_j \ldots$	$\dot{\rightsquigarrow}\ \mathbf{let}\, \overrightarrow{x_i : \tau_i = e_i}\, \mathbf{in}\, e_j$	
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	$\mathbf{case}\ (\mathbf{let}\, \overrightarrow{x : \tau = u}\, \mathbf{in}\, e_0)\ \mathbf{of}\ \overrightarrow{p_i \to e_i}$	$\dot{\rightsquigarrow}\ \mathbf{let}\, \overrightarrow{x : \tau = u}\, \mathbf{in}\ (\mathbf{case}\, e_0\, \mathbf{of}\ \overrightarrow{p_i \to e_i})$	
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	$\mathbf{case}\, e_0\, \mathbf{of}\, \overrightarrow{p_i \to e_i}$	$\dot{\rightsquigarrow}\ \mathbf{case}\, e_0'\, \mathbf{of}\, \overrightarrow{p_i \to e_i}$	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

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 rule 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 only when in an elimination position and not all inlinings. The reason for this is that we want to memoize code that would have triggered ELIMUND and not every intermediate evaluation in the $e \dot{\rightsquigarrow} e'$ judgment.

When MEMOUND fires we also add e to a memoization table and if e ever occurs again in the expression, we replace it with x. We detect reoccurrences of e only when an expression is manifestly equal to e as we use a simple, syntactic comparison. 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. Thus we apply 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 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 $\vdash^{\mathrm{co}} \gamma_1 : \tau_1 \sim \tau_2$ and $\vdash^{\mathrm{co}} \gamma_2 : \tau_2 \sim \tau_1$.

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

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`

CastRefl	$e \triangleright \gamma$	$\rightarrowtail e$ if $\vdash^{\text{co}} \gamma : \tau \sim \tau$
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

Figure 7. Simplifications

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

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

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

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

Figure 8. Cast Symmetry Rules

operator is implemented in terms of `fingerprintFingerprints`, which computes unique hashes for `TypeRep` objects. Furthermore, the 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.

4.5 Optional optimizations

While not essential to the core optimization and the elimination of 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 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 not small, they can represent a significant amount of extra work to be done by the optimization. It would be better to detect whichever branch is dead and skip the extra work in that branch. To do this we apply the CaseBeta rule whenever possible. This rule never makes the code worse or worsens the optimization result. Note that our use of this rule differs from our use of the rules in Figure 6 since we apply this rule at any position in the expression regardless of whether it eliminates an undesirable type.

4.5.2 Memoization floating

If a type occurs in two different places in a type, the memoization may result in duplicated code as the two memoizations may not be in the same scope as each other. 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 `lambda` expressions and the part of the code for `let` expressions. Since neither of these is within the scope of the other, the memoization rule will result in creating a fresh memoization of the traversal over identifiers for each expression even though the code for these two 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 syntax tree 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 `lambda` 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` bindings when possible. 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 a some later point.

4.6 Algorithm summary

The high-level algorithm can be summarized as follows. 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 undesirable types.

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.*

PRIMFF	`fingerprintFingerprints` e	$\dot{\rightsquigarrow}$ $[\![$`fingerprintFingerprints` $e]\!]$	if e is a value
PRIMFFARG	`fingerprintFingerprints` e	$\dot{\rightsquigarrow}$ `fingerprintFingerprints` e'	if $e \ \ddot{\rightsquigarrow} \ e'$
PRIMEQWORDARG1	`eqWord#` $e_1\ e_2$	$\dot{\rightsquigarrow}$ `eqWord#` $e_1'\ e_2$	if $e_1 \ \ddot{\rightsquigarrow} \ e_1'$
PRIMEQWORDARG2	`eqWord#` $e_1\ e_2$	$\dot{\rightsquigarrow}$ `eqWord#` $e_1\ e_2'$	if $e_2 \ \ddot{\rightsquigarrow} \ e_2'$
PRIMEQWORDTRUE	`eqWord#` $l_1\ l_2$	$\dot{\rightsquigarrow}$ `True`	if $l_1 = l_2$
PRIMEQWORDFALSE	`eqWord#` $l_1\ l_2$	$\dot{\rightsquigarrow}$ `False`	if $l_1 \neq l_2$
FORCEDEEP	e	$\ddot{\rightsquigarrow}\ e'$	if $e \ \dot{\rightsquigarrow} \ e'$
FORCEDEEPARG	$e_1\ e_2$	$\ddot{\rightsquigarrow}\ e_1\ e_2'$	if $e_2 \ \ddot{\rightsquigarrow} \ e_2'$

Figure 9. Rules for Primitives

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 occurrence of an undesirable type using the rules from Figure 5.*
   (b) *Eliminate an occurrence of an undesirable type using the rules from Figure 4.*

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.

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 intuition about the transformations necessary, and was then extended with new primitive transformations implementing the rules given in Section 4. The overall optimization algorithm (Algorithm 1) is expressed as a HERMIT script.

HERMIT provides several facilities to ease the implementation of `Core`-to-`Core` transformations such as our optimization. This includes KURE, a strategic rewriting DSL 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 has to be modified to ensure a fair comparison.

Map Found in Magalhães et al. [2010], `Map` performs a mapping over a structure. This traversal is performed on two data types. The first is a binary tree of integers in which all integers are incremented. The second is a logic formula in which all characters in variable names are replaced with the character '`y`'. It is implemented in SYB using `everywhere` and `mkT`.

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 example was implemented both non-generically (Hand) and using SYB combinators (SYB). The SYB implementation was also optimized using 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 2.4 GHz, 64-bit Xeon with 48 GB of RAM running Red Hat Enterprise Linux 6.

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] Due to a space leak in GHC discovered during the course of this work, the latest development version of GHC (as of March 24, 2013) is required to run the HERMIT transformations involved in this optimization.

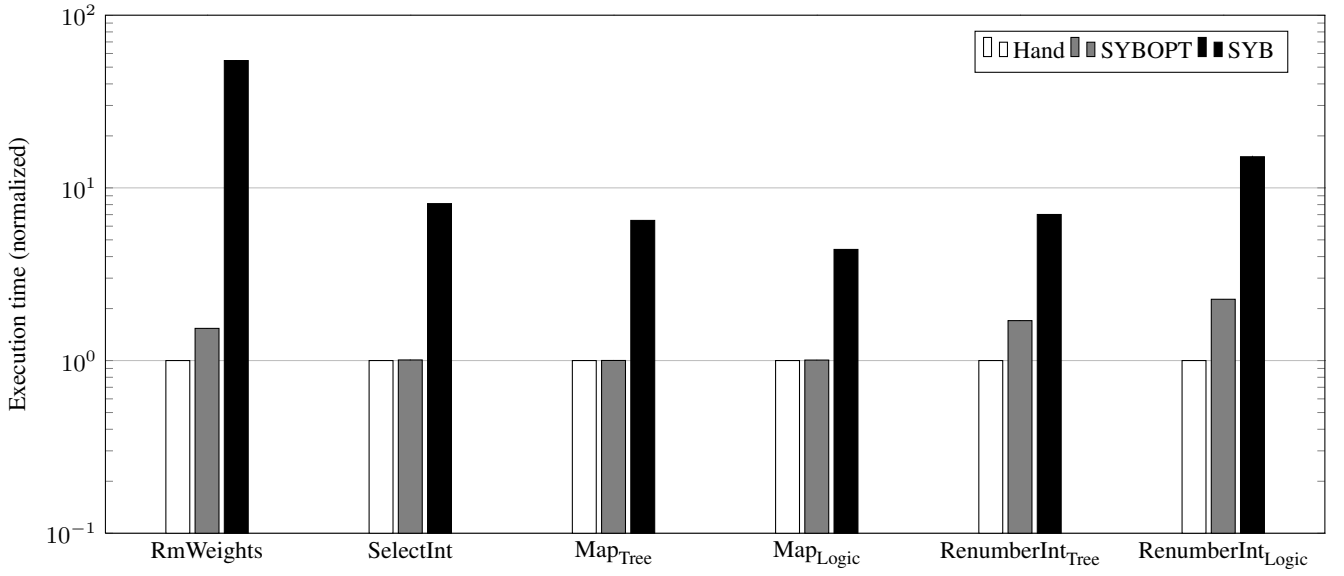

Figure 10. Benchmarks Results

5.3 Performance results

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

With the `SelectInt`, $\texttt{Map}_{Tree}$ and $\texttt{Map}_{Logic}$ benchmarks the optimization completely eliminates the runtime costs associated with SYB. A manual inspection of the generated `Core` confirms that the code produced by the optimization is equivalent to the handwritten code. When initially running these benchmarks, the SYBOPT versions of $\texttt{Map}_{Tree}$ and $\texttt{Map}_{Logic}$ 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] to the code improved its performance to match that of the SYBOPT version.

The SYBOPT version of `RmWeights` performs one-and-a-half times slower than the Hand version. An examination of the optimized `Core` for `RmWeights` reveals why. In SYB, the traversal is specified separately from the transformation to be performed at each point in the traversal. In this benchmark, `everywhere` specifies the traversal while `mkT` specifies the transformation. Both of these inherently involve scrutinizing the constructor of the current node. Thus, in the SYBOPT version each constructor in the object is scrutinized twice, whereas in the Hand version each constructor is scrutinized only once. Inlining the weight removal function generated by our optimization allows GHC to recognize this duplicated work. This eliminates this overhead and on these benchmarks makes SYBOPT as fast as Hand. As this is not an automatic part of our optimization, the results in Figure 10 do not use this inlining, but it does suggest future directions for improvement.

The SYBOPT versions of $\texttt{RenumberInt}_{Tree}$ and $\texttt{RenumberInt}_{Logic}$ are 1.7 and 2.3 times slower than the Hand versions. 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.

In all of the benchmarks, manual inspection of the generated code reveals that the optimization does indeed eliminate the runtime type checks and class-dictionary dispatches from the SYB-style code.

6. Limitations

While the algorithm described in Section 4 is effective for most instances of SYB-style code, it does have limitations. It is a domain-specific optimization and may not be appropriate for all code. In some cases, the compiler may require assistance from the programmer in the form of pragmas or annotations to determine when to apply this optimization.

6.1 Missing inlining information

The first and most obvious limitation is that this optimization relies heavily on inlining, and thus it 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 all occurrences of 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 the 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` instances are underspecified. For example consider the following code.

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

Since this function is polymorphic in `a`, there is no concrete dictionary available for the class constraint `Typeable a`.

Note, however, that this is a problem only at the initial expression being optimized. For example, consider a definition of `mapSYB` like the following.

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

Because this definition of $\texttt{map}_{\texttt{SYB/Int}}$ 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 $\texttt{map}_{\texttt{SYB/Int}}$ even though it would fail on $\texttt{map}_{\texttt{SYB}}$.

6.2 Essential occurrences of undesirable types

Since the primary design heuristic behind this optimization is the elimination of undesirable types, it will run into trouble if there are undesirable types that should not be eliminated. An obvious example is when the type being traversed itself contains undesirable types, 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 function 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 initial code with information about what 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 of an undesirable type and whether it is genuinely undesirable.

6.3 Polymorphic recursion

As with other forms of partial evaluation, polymorphic recursion is a concern with this optimization. Similar to essential occurrences of undesirable types, these polymorphic recursions can be caused either by the type being traversed or by the structure of the traversal. 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 `Double` 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 `Double` has type `T ((Int, Int), (Int, Int))`. 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. We discuss potential future work for handling such types in Section 7.

Alternatively, the traversal itself may be polymorphically recursive in an argument that contains undesirable types. Traversals like this are rare in SYB-style code, but one could imagine an example of this 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 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 contain an undesirable type can freely vary from call to call as the memoization does not care about them.

In the most general case, handling polymorphic recursion is an area of active research in the partial evaluation community. We do not have a general solution to this problem, but in Section 7 we suggest directions for future work that would solve this problem for typical SYB-style code.

7. Future work

The optimization presented in this paper works well when the structure of the generic, SYB-style code closely parallels the structure that we would expect from handwritten code. In some cases, however, the structure dictated by the generic-programming system may not be appropriate for a non-generic traversal.

An instance of this 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 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 any 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.

Another case where the structures of generic and handwritten code differ is traversals over polymorphic types. Most types in Haskell programs are regular, but non-regular, polymorphically recursive types do occasionally occur. As discussed in Section 6.3, the optimization algorithm diverges on these types as it attempts to memoize the generic traversal at an infinite number of types. This limitation is not merely due to a failure of the memoization process. In many cases a non-generic traversal over a polymorphic type must be structured differently than a generic traversal, as it would be 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` where `T` is as defined in Section 6.3. The generic code for this is the following.

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

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

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 = incT' inc x where
  incT' :: (a -> a) -> T a -> T a
  incT' f (Base x) = f x
  incT' f (Double t) = Double (incT' (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 follows the structure of the generic traversal and $\texttt{increment}_\texttt{T}$ does not follow that structure, it is unsurprising that our optimization fails at optimizing such a traversal. However, note that the `f` argument to `incT'` 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 it may be possible to derive such a non-generic implementation from the generic traversal by appropriately specializing and simplifying the `Data` dictionary.

8. Related work

Generic-programming systems in Haskell are often slow relative to handwritten code. There has been a significant amount of work on designing generic programming systems with improved performance [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. 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. Like Bolingbroke and Peyton Jones [2010], we implement a memoization scheme to ensure terms are optimized only once, but, given that we restrict ourselves to optimizing SYB-style code, we can more easily direct the optimization. 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.

9. Conclusion

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

This optimization can be viewed as a form of partial evaluation or supercompilation so its general approach is not new. Because of the domain specific knowledge, however, we can more precisely direct the optimization and avoid issues that arise in more general-purpose optimizations.

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 second author is partially supported by National Science Foundation (NSF) grant number 1117569. The last author is funded by Engineering and Physical Sciences Research Council (EPSRC) grant number 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 Haskell symposium*, Haskell '12, pages 13–24. ACM, 2012. doi: 10.1145/2364506.2364509.

Artem Alimarine and Sjaak Smetsers. Optimizing generic functions. In Dexter Kozen and Carron Shankland, editors, *Mathematics of Program Construction, 7th International Conference*, volume 3125, pages 16–31. Springer, 2004. 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*, Haskell '10, pages 135–146. ACM, 2010. 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*, 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.

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 *Proceedings of the IFIP TC 2 Working Conference on Domain-Specific Languages*, DSL '09, pages 285–309. Springer-Verlag, July 2009. ISBN 978-3-642-03033-8. doi: 10.1007/978-3-642-03034-5_14. URL `http://dx.doi.org/10.1007/978-3-642-03034-5_14`.

Mark P. Jones. Dictionary-free overloading by partial evaluation. *Lisp Symb. Comput.*, 8(3):229–248, September 1995. ISSN 0892-4635. 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 9780130202499.

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. ACM, 2011. doi: 10.1145/1929501.1929507.

Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic progra mming. In *Proceedings of the 2003 ACM SIGPLAN international workshop on Typ es 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 *Accepted for publication at the 24th Symposium on Implementation and Application of Functional Languages (IFL'12)*, IFL '12, 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*, pages 33–42. ACM, 2010. doi: 10.1145/1706356.1706366.

Neil Mitchell and Colin Runciman. Uniform boilerplate and list processing. In *Proceedings of the ACM SIGPLAN Workshop on Haskell*, pages 49–60. ACM, 2007. 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. ACM, 2007. 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):393–433, 2002. 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, 1995.

Neil Sculthorpe, Andrew Farmer, and Andy Gill. The HERMIT in the tree: Mechanizing program transformations in the GHC core language. In *Proceedings of the 24th Symposium on Implementation and Application of Functional Languages*, 2013a. URL `http://www.ittc.ku.edu/csdl/fpg/files/Sculthorpe-13-HERMITinTree.pdf`.

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.

Michael Snoyman. Hackage dependency monitor: Reverse dependency list, 2013. URL `http://packdeps.haskellers.com/reverse`. Accessed on March 26, 2013.

Dimitrios Vytiniotis and Simon Peyton Jones. Practical aspects of evidence-based compilation in System FC. Available at `https://research.microsoft.com/en-us/um/people/simonpj/papers/ext-f/`, 2011.

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.