Optimizing SYB Is Easy!

Michael D. Adams[a,*], Andrew Farmer[b], José Pedro Magalhães[c]

[a]*Department of Computer Science, University of Illinois at Urbana-Champaign*
[b]*Information and Telecommunication Technology Center, University of Kansas*
[c]*Department of Computer Science, University of Oxford*

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.

Keywords: optimization, partial evaluation, datatype-generic programming, Haskell, Scrap Your Boilerplate (SYB), performance
2010 MSC: 68N20

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

*Corresponding author
Email addresses: `afarmer@ittc.ku.edu` (Andrew Farmer), `jpm@cs.ox.ac.uk` (José Pedro Magalhães)
URL: `http://michaeldadams.org` (Michael D. Adams)

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., 2014).

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 was first implemented using HERMIT (Farmer et al., 2012; Sculthorpe et al., 2013), an interactive optimization system implemented as a GHC plugin. HERMIT makes it easy to quickly develop these sorts of optimizations, as a prelude for improving the GHC optimizer with similar techniques. Afterwards, we used the information gained while developing our HERMIT plugin to improve the GHC optimizer, and obtained equally promising results, but without the need for a HERMIT script. The code for our optimization is available at `https://github.com/xich/hermit-syb` and as the `hermit-syb` package on Hackage.

This paper is a rewritten and extended version of our earlier work (Adams et al., 2014). The implementation of our optimization in GHC (together with the description of the changes necessary to the optimizer and SYB) is entirely new to this version. We have also revisited our HERMIT script and its benchmarks to find and eliminate cases of poor performance.

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. In Section 7, we consider the GHC specializer and how it can be adapted to achieve the same optimization. Finally, we 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 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 focus on the `everywhere` traversal. In addition, since traversals over an `AST` type can be 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. Assuming the `Typeable` instances are correct, this use of `unsafeCoerce` 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.

$$\texttt{gmapT f (C } x_1 \texttt{...} x_n \texttt{) = C (f } x_1 \texttt{) ... (f } x_n \texttt{)}$$

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. (2014) 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`$_{\mathrm{SYB}}$ 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 $\mathtt{increment}_{\mathtt{SYB}}$ 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 $\mathtt{increment}_{\mathtt{SYB}}$, we can exploit the fact that, due to the types of $\mathtt{increment}_{\mathtt{SYB}}$ 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 $\mathtt{increment}_{\mathtt{SYB}}$ 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
        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
```

At this point, the outer `mkT` has gone away completely. This is to be expected as `mkT` applied `inc` to only `Int` values, but at the outer level it is being applied to a `[Int]` value in which case `mkT` is an identity.

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

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

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

Figure 1: Syntax of System F_C (Excerpt)

$$
\begin{array}{lll}
\textsc{Beta} & (\lambda x : \tau .\, e_1)\ e_2 & \rightsquigarrow e_1\, [e_2 / x] \\
\textsc{TyBeta} & (\Lambda a : \kappa .\, e)\ \tau & \rightsquigarrow e\, [\tau / a] \\
\textsc{CaseBeta} & \mathbf{case}\, K\, \vec{e_i}\, \mathbf{of}\ \ldots K\, \overrightarrow{x_i : \tau_i} \rightarrow e_j \ldots & \rightsquigarrow e_j \left[\overrightarrow{e_i / x_i} \right] \\
\textsc{Push} & (e_1 \triangleright \gamma)\ e_2 & \rightsquigarrow (e_1\ (e_2 \triangleright \mathbf{sym}\ (\mathbf{nth}\, 1\, \gamma))) \\
& & \qquad \triangleright (\mathbf{nth}\, 2\, \gamma) \\
\textsc{TyPush} & (e \triangleright \gamma)\ \tau & \rightsquigarrow (e\, \tau) \triangleright (\gamma @ \tau)
\end{array}
$$

Figure 2: Reductions of System F_C (Excerpt)

4. A more principled attempt

The transformation in Section 3 is achieved by a simple combination of manually selected 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 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. The judgments used by our optimization are listed in Figure 3 and defined in the following figures.

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 8 as discussed in Section 4.3.*
3. *Evaluate any primitive call using the rules from Figure 10 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 6.*
    (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

$e : \tau$	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, 6 and 8)
$e \xhookrightarrow{\gamma} e'$	Symmetric cast elimination (See Figure 9)
$e \dot{\rightsquigarrow} e'$	Force step (See Figures 7 and 10)
$e \ddot{\rightsquigarrow} e'$	Deep force step (See Figure 10)
$\mathbf{Und}\, \tau$	Undesirable type
$\mathbf{ElimUnd}\, e$	Elimination expression (See Figure 4)
$\mathbf{Memo}\, e$	Memoizable expression (See Figure 6)

Figure 3: Judgments

that they reduce the work that the optimization has to do but are not essential for eliminating expressions that have undesirable types.

With the benchmarks in Section 5 we show that this algorithm successfully optimizes typical SYB-style code to be as fast as handwritten 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 $\texttt{increment}_{\texttt{SYB}}$. 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 by `mkT` and similar functions. 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. For example, we inlined calls to `everywhere`, `mkT`, and `gmapT`, which all take `Data` dictionaries as argument. We also inlined and simplified the call to `cast`

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

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

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

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

Figure 4: Undesirably Typed Expression Elimination

$$\frac{\tau \in \left\{ \begin{array}{c} \texttt{Data}, \texttt{Typeable}, \texttt{TypeRep}, \texttt{TyCon}, \\ \texttt{Fingerprint}, \texttt{ID}, \texttt{CONST}, \texttt{Qi}, \texttt{Qr}, \texttt{Mp} \end{array} \right\}}{\mathbf{Und}\, \tau}$$

$$\frac{\mathbf{Und}\, \tau_1}{\mathbf{Und}\ (\tau_1\, \tau_2)} \qquad \frac{\mathbf{Und}\, \tau_2}{\mathbf{Und}\ (\tau_1\, \tau_2)} \qquad \frac{\mathbf{Und}\, \tau_1}{\mathbf{Und}\ (\tau_1 \to \tau_2)}$$

$$\frac{\mathbf{Und}\, \tau_2}{\mathbf{Und}\ (\tau_1 \to \tau_2)} \qquad \frac{\mathbf{Und}\, \tau}{\mathbf{Und}\ (\forall x : \kappa . \tau)}$$

Figure 5: Undesirable Types

which had `Typeable` dictionaries as arguments. This exposed `TypeRep` objects in the scrutinee of a `case` that we then also symbolically evaluated. 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 7. 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 and is defined in

Figure 5. In addition, we use typing judgments for expressions, $e : \tau$, and 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 7 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 rules ForceAppFun, ForceAppTyFun, ForceScr, ForceLetBody, and ForceCast 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.

In the example in Section 3, we already have a binding for such an expression, namely $\texttt{increment}_{\texttt{SYB}}$. In general, however, we cannot rely on such a binding already being in scope. The original call to `everywhere` might be embedded in another expression, or `everywhere` might be called over a non-uniform type for abstract syntax that contains mutually recursive types for expressions and statements. Even if there is a binding for the type at which `everywhere` is originally called, we need bindings for the other types. Since we cannot rely on the existence of these bindings, we introduce them when we first start simplifying an expression for which we might later need a binding.

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 6. This rule has higher priority than ElimUnd and should be used instead of that rule whenever possible. In Section 3, the applications of `everywhere`, `mkT`, `gmapT`, and `cast` would all be memoized under this rule. This strategy may lead to unnecessary extra memoization bindings. However, this heuristic is easy to implement, and the extra bindings do not get in the way of the rest of the optimization.

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

$$\frac{\textbf{Memo}\, e_1}{\textbf{Memo}\ (e_1\, e_2)}\ \textsc{MemoUndApp} \qquad \frac{\textbf{Memo}\, e_1}{\textbf{Memo}\ (e_1\, \tau)}\ \textsc{MemoUndTyApp}$$

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

Figure 6: Undesirably Typed Expression Memoization

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 6. 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 8 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 `newtype` definitions and some class dictionaries makes use of casts. For example, a call to the `typeOf` method of the `Typeable` class is implemented as a cast. In addition, many of the SYB functions use

ForceBeta

$$(\lambda x : \tau .\, e_1)\ e_2 \quad \dot{\rightsquigarrow}\ \mathbf{let}\, x : \tau = e_2 \,\mathbf{in}\, e_1$$

ForceTyBeta

$$(\Lambda a : \kappa .\, e)\ \tau \quad \dot{\rightsquigarrow}\ \mathbf{let}\, a : \kappa = \tau \,\mathbf{in}\, e$$

ForceCaseBeta

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

ForcePush

$$(e_1 \triangleright \gamma)\ e_2 \quad \dot{\rightsquigarrow}\ (e_1\ (e_2 \triangleright \mathbf{sym}\ (\mathbf{nth}\, 1\, \gamma))) \triangleright (\mathbf{nth}\, 2\, \gamma)$$

ForceTyPush

$$(e \triangleright \gamma)\ \tau \quad \dot{\rightsquigarrow}\ (e\, \tau) \triangleright (\gamma @ \tau)$$

ForceVar

$$x \quad \dot{\rightsquigarrow}\ e$$

if e is the inlining of x

ForceLetFloatApp

$$(\mathbf{let}\, \overrightarrow{x : \tau = e_i}\, \mathbf{in}\, e_0)\ u \quad \dot{\rightsquigarrow}\ \mathbf{let}\, \overrightarrow{x : \tau = e_i}\, \mathbf{in}\, e_0\, u$$

ForceLetFloatScr

$$\mathbf{case}\ (\mathbf{let}\, \overrightarrow{x : \tau = u}\, \mathbf{in}\, e_0)\ \mathbf{of}\, \overrightarrow{p_i \to e_i} \quad \dot{\rightsquigarrow}\ \mathbf{let}\, \overrightarrow{x : \tau = u}\, \mathbf{in}\ (\mathbf{case}\, e_0\, \mathbf{of}\, \overrightarrow{p_i \to e_i})$$

ForceAppFun

$$e_1\, e_2 \quad \dot{\rightsquigarrow}\ e_1'\, e_2$$

if $e_1 \dot{\rightsquigarrow} e_1'$

ForceAppTyFun

$$e_1\, \tau \quad \dot{\rightsquigarrow}\ e_1'\, \tau$$

if $e_1 \dot{\rightsquigarrow} e_1'$

ForceScr

$$\mathbf{case}\, e_0\, \mathbf{of}\, \overrightarrow{p_i \to e_i} \quad \dot{\rightsquigarrow}\ \mathbf{case}\, e_0'\, \mathbf{of}\, \overrightarrow{p_i \to e_i}$$

if $e_0 \dot{\rightsquigarrow} e_0'$

ForceLetBody

$$\mathbf{let}\, \overrightarrow{x_i : \tau_i = u_i}\, \mathbf{in}\, e \quad \dot{\rightsquigarrow}\ \mathbf{let}\, \overrightarrow{x_i : \tau_i = u_i}\, \mathbf{in}\, e'$$

if $e_0 \dot{\rightsquigarrow} e_0'$

ForceCast

$$e \triangleright \gamma \quad \dot{\rightsquigarrow}\ e' \triangleright \gamma$$

if $e \dot{\rightsquigarrow} e'$

Figure 7: Forcing Rules

$$
\begin{array}{lllll}
\textsc{CastRefl} & e \triangleright \gamma & & \rightarrowtail e & \text{if } \gamma : \tau \sim \tau \\
\textsc{CastSym} & e \triangleright \gamma & & \rightarrowtail e' & \text{if } e \overset{\gamma}{\hookrightarrow} e' \\
\textsc{DeadLet} & \mathbf{let}\, x : \tau & = u & \mathbf{in}\, e \rightarrowtail e & \text{if } x \notin fv\,(e) \text{ and } x \\
 & & & & \text{is not a memoization} \\
\textsc{SubstStar} & \mathbf{let}\, x : \star & = \tau & \mathbf{in}\, e \rightarrowtail e\,[\tau & /x] \\
\textsc{SubstHash} & \mathbf{let}\, x : \# & = \tau & \mathbf{in}\, e \rightarrowtail e\,[\tau & /x] \\
\textsc{SubstVar} & \mathbf{let}\, x : \tau & = x' & \mathbf{in}\, e \rightarrowtail e\,[x' & /x] \\
\textsc{SubstLit} & \mathbf{let}\, x : \tau & = l & \mathbf{in}\, e \rightarrowtail e\,[l & /x] \\
\textsc{SubstDFun} & \mathbf{let}\, x : \tau & = v\,\vec{u} & \mathbf{in}\, e \rightarrowtail e\,[v\,\vec{u}/x] & \text{if } v \text{ is a} \\
 & & & & \text{dictionary constructor}
\end{array}
$$

Figure 8: Simplifications

`newtype` definitions to define higher level operations in terms of lower level operations. For example, the default implementation of `gmapT` is in terms of `gfoldl` with `ID` for the `c` type parameter. Since `ID` is a `newtype`, calls to the `ID` constructor and `unID` destructor get translated to casts.

As these higher-level operations project into and out of these types, these constructors and destructors may be directly or indirectly nested on each other as pairs of symmetric of casts that could be eliminated. In addition, the types in these casts may be refined by ForceTyBeta or shuffled around by the ForcePush or ForceTyPush rules to result in casts that are reflexive.

These casts can quickly build up and get in the way of the core optimization rules. For example, it often happens that the scrutinee of a `case` contains 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.

Reflexive casts from a type to itself are directly eliminated with the CastRefl rule, which just checks the type of the cast. Symmetric casts, however, could be separated from each other by intermediate forms as in the following example where $\gamma_1 : \tau_1 \sim \tau_2$, $\gamma_2 : \tau_2 \sim \tau_1$, and $\gamma_3 : \tau_2 \sim \tau_1$.

$$(\mathbf{case}\, x\, \mathbf{of}\ \{C_1 \rightarrow e_1 \triangleright \gamma_2; C_2 \rightarrow e_2 \triangleright \gamma_3\}) \triangleright \gamma_1$$

Simplifying this expression is accomplished by the CastSym rule. This rule uses the $e \overset{\gamma}{\hookrightarrow} e'$ judgment in Figure 9 to check whether all paths down e contain a cast symmetric to γ. That judgment returns the expression with those symmetric casts removed as e', and thus CastSym reduces our example to to the following.

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

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

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

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

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

Figure 9: Cast Symmetry Rules

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 8. 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#` and `tagToEnum#` primitives. 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 10 where we use double brackets ($[\![$ and $]\!]$) for compile time evaluation. These rules effectively implement constant folding for these operators.

PrimFF

$$\begin{array}{ll} \texttt{fingerprintFingerprints}\, e & \dot{\rightsquigarrow} [\![\texttt{fingerprintFingerprints}\, e]\!] \\ & \quad \text{if } e \text{ is a value} \end{array}$$

PrimFFArg

$$\begin{array}{ll} \texttt{fingerprintFingerprints}\, e & \dot{\rightsquigarrow} \texttt{fingerprintFingerprints}\, e' \\ & \quad \text{if } e \ \ddot{\rightsquigarrow} e' \end{array}$$

PrimEqWord

$$\begin{array}{ll} \texttt{eqWord\#}\, e_1\, e_2 & \dot{\rightsquigarrow} [\![\texttt{eqWord\#}\, e_1\, e_2]\!] \\ & \quad \text{if } e_1 \text{ and } e_2 \text{ are values} \end{array}$$

PrimEqWordArg1

$$\begin{array}{ll} \texttt{eqWord\#}\, e_1\, e_2 & \dot{\rightsquigarrow} \texttt{eqWord\#}\, e_1'\, e_2 \\ & \quad \text{if } e_1 \ddot{\rightsquigarrow} e_1' \end{array}$$

PrimEqWordArg2

$$\begin{array}{ll} \texttt{eqWord\#}\, e_1\, e_2 & \dot{\rightsquigarrow} \texttt{eqWord\#}\, e_1\, e_2' \\ & \quad \text{if } e_2 \ddot{\rightsquigarrow} e_2' \end{array}$$

TagToEnum

$$\begin{array}{ll} \texttt{tagToEnum\#}\, e_1 & \dot{\rightsquigarrow} [\![\texttt{tagToEnum\#}\, e_1]\!] \\ & \quad \text{if } e \text{ is a value} \end{array}$$

TagToEnumArg

$$\begin{array}{ll} \texttt{tagToEnum\#}\, e_1 & \dot{\rightsquigarrow} \texttt{tagToEnum\#}\, e_1' \\ & \quad \text{if } e \ \ddot{\rightsquigarrow} e' \end{array}$$

ForceDeep

$$\begin{array}{ll} e & \ddot{\rightsquigarrow} e' \\ & \quad \text{if } e \ \dot{\rightsquigarrow} e' \end{array}$$

ForceDeepArg

$$\begin{array}{ll} e_1\, e_2 & \ddot{\rightsquigarrow} e_1\, e_2' \\ & \quad \text{if } e_2 \ddot{\rightsquigarrow} e_2' \end{array}$$

Figure 10: Rules for Primitives

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 the purpose of `mkT`. 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 7 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 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., 2013). 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 in Algorithm 1 was implemented as a HERMIT plugin. After the optimization completes, we use HERMIT's `simplify` command to perform basic simplification like dead `let`-binding elimination.

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., 2014), 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 450 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 two[1] data types. The first is a binary tree of integers. The second is a logic formula. For the binary tree, all integers are incremented. For the other type, 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 (SYB/Hermit). 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 7.8.2 using the `-O2` compiler option and ran them with the `-K1g` RTS option on a 1.7 GHz, 64-bit Intel i7 with 8 GB of RAM running Darwin 13.1.0.

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. Similiary, the Hand implementation of `RenumberInt` in `GPBench` did not descend into strings, whereas the SYB implementation does. We modifed the Hand implementation to match the SYB traversal.

5.3. Performance results

Figure 11 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.

[1] Note to reviewers: Due to a combination of lost files and newly discovered bugs in GHC and/or HERMIT, we are currently unable to present the $\texttt{Map}_{\texttt{Src}}$ benchmark results that were in the previous version of this paper. We are working on these problems and hope to add these results back into the paper once these issues are solved.

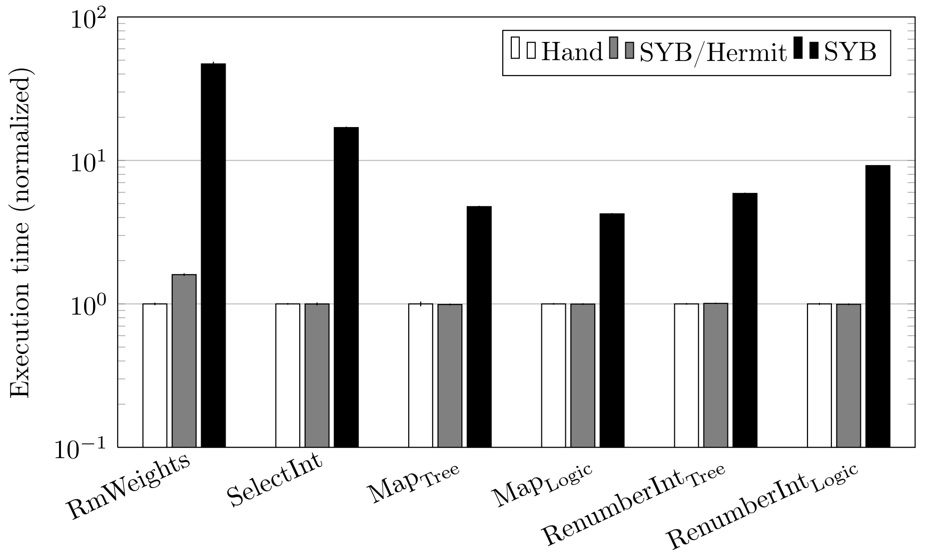

Figure 11: Benchmarks Results

For all of the benchmarks except `RmWeights`, the optimization completely eliminates the runtime costs associated with SYB. A manual inspection of the generated `Core` confirms that the optimization does indeed eliminate all run-time type checks and dictionary dispatches in the SYB-style code and that the resulting code is equivalent to the handwritten code.

When initially running these benchmarks, the SYB/Hermit 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 SYB/Hermit version.

On the `RmWeights` benchmark, SYB/Hermit fails to achieve parity with the Hand version. This contrasts with a previous version of this paper (Adams et al., 2014) in which `RmWeights` fully optimized. Inspecting the `Core` reveals why. The optimization successfully eliminates all runtime type checks and dictionary dispatches as expected. After several of GHC's own optimization passes run, including two rounds of the simplifier, we are left with the following two mutually-recursive functions. (Casts have been omitted for clarity.)

```
memo_everywhere :: WTree Int Int → WTree Int Int
memo_everywhere = λ x →
  case memo_gfoldl x of
   WithWeight t w → t
   wild → wild
```

```
memo_gfoldl :: WTree Int Int → WTree Int Int
memo_gfoldl = λ ds →
  case ds of
    Leaf a1 → Leaf Int Int a1
    Fork a1 a2 →
      Fork Int Int (memo_everywhere a1) (memo_everywhere a2)
    WithWeight a1 a2 →
      WithWeight Int Int (memo_everywhere a1) a2
```

In order to achieve the same performance as Hand, the `memo_gfoldl` function needs to be inlined into `memo_everywhere`. This causes a subsequent `case`-of-`case` transformation and `case`-reduction by the simplifier, resulting in a single self-recursive traversal function. When we tested this by forcing the inlining with HERMIT, we observe the desired speedup. However, GHC marks `memo_gfoldl` as a loop breaker, an annotation it uses to ensure that inlinings in mutually recursive binding groups terminate. This prevents the full optimization of this code. We speculate that `RmWeights` fully optimized in the previous version of this paper because a different function was chosen by GHC as the loop breaker. However, we have no way to test this as we no longer have the particular development version of GHC used in that paper.

In the previous version of this paper, $\texttt{RenumberInt}_{Logic}$ performed 2.2 times slower than the Hand version. Subsequent investigation has revealed the cause. Part of this slowdown was due to the selective traversal issue we mention in Section 5.2, namely, the SYB version descended into strings while the Hand version did not. Fixing this brought SYB/Hermit to 1.6 times slower than Hand. Investigating the resulting `Core` showed that the remaining slowdown was caused by poor interactions with GHC's unboxing optimizations.

Recall that the `RenumberInt` benchmark uses a `State` monad to generate fresh integers during the traversal. The `State` monad in Haskell is implemented using a function which returns a tuple of value and state. Combining two `State` computations with `>>=` results in the allocation of a tuple for the result of the first computation followed by a `case` expression to extract the value and state from the tuple for use by the second computation. This intermediate allocation of tuples is wasteful so, when possible, GHC's Constructed Product Result (CPR) analysis pass (Baker-Finch et al., 2004) eliminates tuples by unboxing.

The code resulting from our optimization prevents this unboxing. We speculate that residual casts are interfering with the CPR analysis. We can improve the situation by switching to the strict `State` monad, which immediately scrutinizes the result of the first `State` computation rather than allocating it with a `let` binding. This makes the code resulting from our optimization amenable to CPR, which then successfully unboxes the tuples. Switching to a strict `State` monad for $\texttt{RenumberInt}_{Tree}$ and $\texttt{RenumberInt}_{Logic}$ improves the running time of Hand by a factor of 1.1, the unoptimized SYB by a factor of 1.2, and SYB/Hermit by a factor of 1.8 at which point SYB/Hermit matches the performance of Hand. The results for $\texttt{RenumberInt}_{Tree}$ and $\texttt{RenumberInt}_{Logic}$ in Figure 11 are for the strict `State` monad.

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 issue 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. For example, by default, the inlining information for several operations in SYB were not available so we had to use `-fexpose-all-unfoldings` or add `INLINABLE` pragmas to expose these. Missing inlining information 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 one might write as a helper function.

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

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 `map`$_{\mathtt{SYB}}$ 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 `map`$_{\mathtt{SYB}}$, we optimize uses of `map`$_{\mathtt{SYB}}$ such as the following.

```
increment_{SYB/Int} :: [Int] -> [Int]
increment_{SYB/Int} = map_{SYB} 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 `Data` 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 $\texttt{map}_{\texttt{SYB}}$ that are successfully optimized by our optimization can be explicitly generated by specifying their types as in the following.

```
map_{SYB/Int} :: (Int -> Int) -> [Int] -> [Int]
map_{SYB/Int} = map_{SYB}
```

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 have undesirable types but 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 from Hinze et al. (2006).

```
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.

---

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

```
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.

```
increment_T :: T Int -> T Int
increment_T 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.

```
increment_T :: T Int -> T Int
increment_T 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 `increment`$_{T}$ 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.

As with polymorphically recursive types, this limitation is not unique to optimizing SYB-style code. Polymorphic recursion is an area of active research in the partial evaluation community for which we do not have a solution in the general case.

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

$$
\begin{array}{lll}
\mathtt{memo}_{\mathtt{Char}} & \mathtt{c} & = \mathtt{c} \\
\mathtt{memo}_{\mathtt{String}} & \mathtt{[]} & = \mathtt{[]} \\
\mathtt{memo}_{\mathtt{String}} & \mathtt{(c : cs)} & = \mathtt{memo}_{\mathtt{Char}}\ \mathtt{c}\ \mathtt{:}\ \mathtt{memo}_{\mathtt{String}}\ \mathtt{cs}
\end{array}
$$

Here $\mathtt{memo}_{\mathtt{String}}$ 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. The GHC Specializer

Given that the core rules of our optimization specialize functions to particular arguments, a natural question is whether the existing specializer in GHC can achieve the same effect. However, the GHC specializer focuses on class-dictionary specialization (Jones, 1995) and does not specialize non-dictionary arguments. This is a problem in `increment`$_{\texttt{SYB}}$ where we need to specialize `everything` over the non-dictionary argument `mkT inc`. As a consequence, the default optimization pipeline in GHC does not do the specialization needed to effectively optimize SYB-style code.

The situation is not a total loss, however. Under appropriate conditions the GHC specializer will specialize some parts of `increment`$_{\texttt{SYB}}$ over the `[Int]` type and produce the following code.

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

everywhere_[Int] :: (∀b. Data b => b -> b)
                 -> [Int] -> [Int]
everywhere_[Int] f x = f (gmapT_[Int] (everywhere f) x)

gmapT_[Int] :: (∀b. Data b => b -> b)
            -> [Int] -> ID [Int]
gmapT_[Int] f [] = []
gmapT_[Int] f (x : xs) = f x : f xs
```

Unfortunately, while `everywhere` and `gmapT` are specialized to particular types in this code, the `f` arguments to these functions are not. They are still polymorphic and take class dictionaries as arguments. This is because the techniques used by the GHC specializer do not handle the rank-2 polymorphism of these arguments. As a consequence, when `increment`$_{\texttt{SYB}}$ is invoked, the outermost call to `everywhere` and `gmapT` use the specialized version, but the inner calls to `everywhere` that are made by `gmapT` use the unspecialized version. As a result, the bulk of the computation runs slowly and does not use these specialized versions of `everywhere` and `gmapT`.

Even though the default GHC optimization pipeline does not do well on this code, there are some things we can do to help it. First, we can manually perform a static argument transformation on `everywhere` and define it as follows.

```
everywhere :: (∀b. Data b => b -> b)
```

```
                 -> (∀a. Data a => a -> a)
      everywhere f x = go x where
        go :: ∀c. Data c => c -> c
        go x = f (gmapT go x)
```

With this definition, inlining `everywhere` produces a version of `go` that implements the work of `everywhere` but specialized to one particular value of `f`. For example, inlining `everywhere` into `increment`$_{\texttt{SYB}}$ results in the following.

```
      increment_SYB :: [Int] -> [Int]
      increment_SYB x = go where
        go :: ∀c. Data c => c -> c
        go x = mkT inc (gmapT go x)
```

The resulting `go` function implements `everywhere` but specialized to `mkT inc` for `f`. More importantly though, `go` does not involve any higher-rank polymorphism. Thus if we run the specializer on this code, we get the following which contains a version of `go` specialized to the `[Int]` type.

```
      increment_SYB :: [Int] -> [Int]
      increment_SYB x = go_[Int] where
        go :: ∀c. Data c => c -> c
        go x = mkT inc (gmapT go x)
        go_[Int] :: [Int] -> [Int]
        go_[Int] x = mkT inc (gmapT go x)
```

In the default GHC optimization pipeline, the specializer runs before the inlining process in the simplifier. Thus, in order to get this code, we have to modify GHC to run the specializer after inlining.

This version is not yet fully optimized, however, as `go`$_{\texttt{[Int]}}$ still contains calls to the polymorphic functions `mkT` and `gmapT`. Since `mkT` is not recursive, inlining and symbolically evaluating should expose the `cast` in `mkT` and then allow us to evaluate the comparison of `TypeRep` objects in the `cast`. That would transform `go`$_{\texttt{[Int]}}$ to the following.

```
      go_[Int] :: [Int] -> [Int]
      go_[Int] x = gmapT go x
```

Unfortunately, in our experiments with GHC the optimization process often simplified `mkT` and the contained `cast` but did not do the final step of removing the comparison over `TypeRep` objects. This seems to be due to the simplifier not knowing how to symbolically evaluate the `fingerprintFingerprints` function that sometimes arises when simplifying such code. Adding primitive simplification rules such as those in Figure 10 is a relatively trivial extension to the GHC optimizer and allows us to eliminate that part of the code.

Next, consider the call to `gmapT` inside `go`$_{\texttt{[Int]}}$. It is also over the concrete type `[Int]` so the simplifier statically computes the dictionary dispatch and changes the code to use `gmapT`$_{\texttt{[Int]}}$, the `gmapT` implementation in the `Data` instance for `[Int]`. This results in the following code.

```
go_[Int] :: [Int] -> [Int]
go_[Int] x = gmapT_[Int] go x
```

Unfortunately, the simplifier does not inline this invocation of $\texttt{gmapT}_{\texttt{[Int]}}$. This is because there is a cycle between the dictionary for `Data` at the `[Int]` type and $\texttt{gmapT}_{\texttt{[Int]}}$, so GHC marks one of them as a loop breaker in order to avoid infinite inlinings. As GHC avoids making class dictionaries be loop breakers, $\texttt{gmapT}_{\texttt{[Int]}}$ is marked. As a result, the simplifier does not inline $\texttt{gmapT}_{\texttt{[Int]}}$. If we overlook this obstacle and force $\texttt{gmapT}_{\texttt{[Int]}}$ to inline, then we get the following code.

```
go_[Int] :: [Int] -> [Int]
go_[Int] [] = []
go_[Int] (x : xs) = go x : go xs
```

This exposes two calls to `go`. One is on `x` and is over the `Int` type. The other is on `xs` and is over the `[Int]` type. When the GHC specializer adds specializations, it also adds rewrite rules for those specializations. The simplifier uses these rewrite rules to convert the call to `go` over the `[Int]` type to $\texttt{go}_{\texttt{[Int]}}$.

At this point, all of the dictionaries involving the `[Int]` type have been removed by the GHC specializer and simplifier. However, note that we still have the call `go x` which is on the `Int` type. In this particular case, the `Data` and `Typeable` instances for `Int` are simple enough that later passes of the GHC optimization pipeline do transform this into `inc x`. However, this is not always the case. Consider, for example, what happens if $\texttt{increment}_{\texttt{SYB}}$ is over `[[Int]]` instead of `[Int]`. After the initial inlining of `everywhere` we end up with the following code.

```
increment_SYB :: [[Int]] -> [[Int]]
increment_SYB x = go where
  go :: ∀c. Data c => c -> c
  go x = mkT inc (gmapT go x)
```

This code contains a manifest call to `go` at the `[[Int]]` type, but there is no such manifest call to `go` at the `[Int]` type. This is because `go` is passed as a polymorphic argument to `gmapT`. This is exactly the sort of argument that the GHC specializer does not know how to handle. With `everywhere` we were able get around this by doing a static argument transformation, but `gmapT` is a class method so we cannot do the same. Thus when the GHC specializer runs, `go` is specialized only at the `[[Int]]` type. This results in the following code.

```
increment_SYB :: [[Int]] -> [[Int]]
increment_SYB x = go_[[Int]] where
  go :: ∀c. Data c => c -> c
  go x = mkT inc (gmapT go x)
  go_[[Int]] :: [[Int]] -> [[Int]]
  go_[[Int]] x = mkT inc (gmapT go x)
```

As before, we simplify away the `mkT` in $\texttt{go}_{\texttt{[[Int]]}}$. Since the `gmapT` in $\texttt{go}_{\texttt{[[Int]]}}$ is over the concrete type `[[Int]]`, we also simplify that, which results in the following for $\texttt{go}_{\texttt{[[Int]]}}$.

```
go_[[Int]] :: [[Int]] -> [[Int]]
go_[[Int]] [] = []
go_[[Int]] (x : xs) = go x : go xs
```

The call `go xs` is over the type `[[Int]]` for which we have a specialization so this is then turned into the following.

```
go_[[Int]] :: [[Int]] -> [[Int]]
go_[[Int]] [] = []
go_[[Int]] (x : xs) = go x : go_[[Int]] xs
```

As before, we have the call `go x` over a type which does not have a specialization. This time, however, it is over the type `[Int]`, which is complicated enough that it will not be optimized by the later stages of the pipeline.

Thus one pass of the specializer is not sufficient to optimize this SYB-style code. However, note that after specialization and simplification, we now have a manifest call of `go` on the `[Int]` type. The GHC specializer knows how to handle this sort of call. Thus if we run the specializer over this code, we get a $\texttt{go}_{\texttt{[Int]}}$ function. After another round of inlining and simplification this results in the following.

```
increment_SYB :: [[Int]] -> [[Int]]
increment_SYB x = go where
  go :: ∀c. Data c => c -> c
  go x = mkT inc (gmapT go x)
  go_[[Int]] :: [[Int]] -> [[Int]]
  go_[[Int]] [] = []
  go_[[Int]] (x : xs) = go_[Int] x : go_[[Int]] xs
  go_[Int] :: [Int] -> [Int]
  go_[Int] [] = []
  go_[Int] (x : xs) = go x : go_[Int] xs
```

This in turn has exposed a call to `go` on the `Int` type inside $\texttt{go}_{\texttt{[Int]}}$. This is simple enough that we can either leave this for later passes in the optimization pipeline or we can invoke the specializer and simplifier again, which results in the following.

```
increment_SYB :: [[Int]] -> [[Int]]
increment_SYB x = go where
  go_[[Int]] :: [[Int]] -> [[Int]]
  go_[[Int]] [] = []
  go_[[Int]] (x : xs) = go_[Int] x : go_[[Int]] xs
  go_[Int] :: [Int] -> [Int]
```

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

Each time we invoke the specializer and then the simplifier, we potentially uncover more types at which to specialize `go`. Thus for complex types we may need to perform this iteration multiple times.

In summary, while the default GHC optimization pipeline does not effectively optimize SYB-style code, a few modifications are sufficient to do so. First, we static-argument transform traversals like `everywhere`. This allows them to be inlined, which effectively specializes them to their first argument. Second, we run the specializer after the simplifier so that the `go` function resulting from the inlining of `everywhere` is specialized to particular types. Third, we run the simplifier again with two modifications. We symbolically evaluate `TypeRep`, `TyCon`, and `Fingerprint` calculations using rules such as those in Figure 10. We also inline the type-specific instances of `gfoldl`, `gmapT`, `gmapQ`, and `gmapM` even though the cycles in these would normally prevent it. Fourth, we iterate this specialization and simplification process with new iterations each time it reveals a new type at which to specialize `go`. The end result of all this is a fully optimized version of the code that contains no `Data` or `Typeable` class dispatches and no `TypeRep`, `TyCon`, or `Fingerprint` computations.

Note that using the specializer in this way is not as general as the optimization described in Section 4. It relies on the code having a structure similar to `everywhere` that we can static-argument transform. Thus, this technique will not be as effective when the code is not or cannot be of such a form.

7.1. Benchmarks

In order to test the efficacy of this technique, we created a plugin for GHC that iterates between specialization and simplification. This plugin also inlined of any instance specific implementations of `gfoldl`, `gmapT`, `gmapQ`, or `gmapM` that the GHC simplifier did not and used the simplification rules in Figure 10 to eliminate `TypeRep`, `TyCon`, and `Fingerprint` computations. We then re-ran the benchmarks in Section 5 with this plugin and a modified version of the SYB library with the static argument transformation applied to all of the traversal schemes.

We ran the benchmarks both with (Spec$^+$) and without (Spec) the extra simplification rules for `TypeRep`, `TyCon`, and `Fingerprint` computations. Also, since these tests involve a different version of SYB than in Section 5, we reran the benchmarks without our optimization or plugin (SYB). The results are plotted in Figure 12 and are normalized relative to the handwritten code (Hand) just as in Figure 11.

Overall, both Spec and Spec$^+$ performed well. Spec$^+$ ran on par with the handwritten code across the board. Spec also ran on par with Hand for most benchmarks, but failed to significantly improve `RmWeights` or `SelectInt`. An inspection of the resulting core from each of these benchmarks reveals why. In

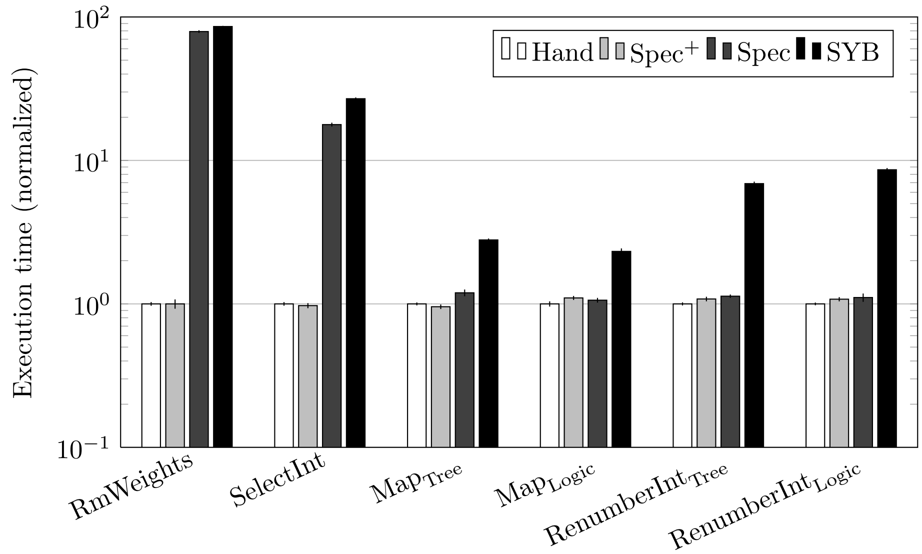

Figure 12: Benchmarks Results

most of the benchmarks, the `TypeRep`, `TyCon`, and `Fingerprint` computations that are left over after our plugin iteratively runs the specializer are simple enough that they are eliminated by later passes of the compiler. In `RmWeights` and `SelectInt`, however, the computations are more complex and are not eliminated.

8. Related work

Generic-programming systems in Haskell are often slow relative to hand-written 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.

Our optimization is related to class dictionary specialization (Jones, 1995) and 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. As discussed in Section 7, dictionary specialization is not sufficient to optimize SYB-style code, but using the lessons and experience from our work we were able to find modifications of the GHC specializer to effectively optimize SYB-style code.

In a broader 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.

Our optimization can also 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. We can draw direct connections to many of the rules in Jonsson and Nordlander (2011). For example, rules R1, R5, R6, R12, and R13 in that work correspond to several of the forcing rules in our Figure 7. Rules R2, R11, and R15 correspond to the primitive simplification rules in Figure 8. Rule R8 and R9 respectively correspond to SubstLit and SubstVar in Figure 8.

However, unlike general partial evaluation, we take advantage of domain knowledge about SYB-style code. In particular, we use the types of expressions to direct the optimization and start symbolically evaluating an expression only when it is a form that eliminates an expression with an undesirable type. In theory, we face the same problem of code explosion that supercompilers do, 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 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 first 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. We have then explored how to integrate this optimization directly into GHC in order to obtain similar results without depending on HERMIT. In the future, we plan to investigate how to make our optimization 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.

Acknowledgements Our thanks go to Simon Peyton Jones for his help with testing the GHC optimizer. 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

Adams, M. D., DuBuisson, T. M., 2012. Template your boilerplate: using Template Haskell for efficient generic programming. In: Proceedings of the 2012 symposium on Haskell symposium. Haskell '12. ACM, New York, NY, USA, pp. 13–24.

Adams, M. D., Farmer, A., Magalhães, J. P., 2014. Optimizing SYB is easy! In: Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation. PEPM '14. ACM, New York, NY, USA, pp. 71–82.

Alimarine, A., Smetsers, S., 2004. Optimizing generic functions. In: Kozen, D. (Ed.), Mathematics of Program Construction. Vol. 3125 of Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 16–31.

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

Baker-Finch, C., Glynn, K., Peyton Jones, S., Mar. 2004. Constructed product result analysis for Haskell. Journal of Functional Programming 14 (02), 211–245.

Bolingbroke, M., Peyton Jones, S., 2010. Supercompilation by evaluation. In: Proceedings of the third ACM Haskell symposium on Haskell. Haskell '10. ACM, New York, NY, USA, pp. 135–146.

Brown, N. C. C., Sampson, A. T., 2009. Alloy: fast generic transformations for Haskell. In: Proceedings of the 2nd ACM SIGPLAN symposium on Haskell. Haskell '09. ACM, New York, NY, USA, pp. 105–116.

Chakravarty, M. M. T., Ditu, G. C., Leshchinskiy, R., 2009. Instant generics: Fast and easy, available at `http://www.cse.unsw.edu.au/~chak/papers/instant-generics.pdf`.

Farmer, A., Gill, A., Komp, E., Sculthorpe, N., 2012. The HERMIT in the machine: a plugin for the interactive transformation of GHC core language programs. In: Proceedings of the 2012 Haskell Symposium. Haskell '12. ACM, New York, NY, USA, pp. 1–12.

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

Gill, A., 2009. A Haskell hosted DSL for writing transformation systems. In: Taha, W. M. (Ed.), Domain-Specific Languages. Vol. 5658 of Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 285–309.

Hinze, R., Löh, A., Oliveira, B. C. d. S., 2006. "scrap your boilerplate" reloaded. In: Hagiya, M., Wadler, P. (Eds.), Functional and Logic Programming. Vol. 3945 of Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 13–29.

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

Jones, M. P., Sep. 1995. Dictionary-free overloading by partial evaluation. LISP and Symbolic Computation 8 (3), 229–248.

Jones, N. D., Gomard, C. K., Sestof, P., 1993. Partial Evaluation and Automatic Program Generation. Prentice-Hall International Series in Computer Science. Prentice Hall.

Jonsson, P. A., Nordlander, J., 2011. Taming code explosion in supercompilation. In: Proceedings of the 20th ACM SIGPLAN workshop on Partial evaluation and program manipulation. PEPM '11. ACM, New York, NY, USA, pp. 33–42.

Lämmel, R., Peyton Jones, S., 2003. 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. ACM, New York, NY, USA, pp. 26–37.

Lämmel, R., Peyton Jones, S., 2004. Scrap more boilerplate: reflection, zips, and generalised casts. In: Proceedings of the ninth ACM SIGPLAN international conference on Functional programming. ICFP '04. ACM, New York, NY, USA, pp. 244–255.

Magalhães, J. P., 2013. Optimisation of generic programs through inlining. In: Implementation and Application of Functional Languages.

Magalhães, J. P., Holdermans, S., Jeuring, J., Löh, A., 2010. Optimizing generics is easy! In: Proceedings of the 2010 ACM SIGPLAN workshop on Partial evaluation and program manipulation. PEPM '10. ACM, New York, NY, USA, pp. 33–42.

Mitchell, N., Runciman, C., 2007. Uniform boilerplate and list processing. In: Proceedings of the ACM SIGPLAN workshop on Haskell workshop. Haskell '07. ACM, New York, NY, USA, pp. 49–60.

Peyton Jones, S., 2007. Call-pattern specialisation for Haskell programs. In: Proceedings of the 12th ACM SIGPLAN international conference on Functional programming. ICFP '07. ACM, New York, NY, USA, pp. 327–337.

Peyton Jones, S., Marlow, S., Jul. 2002. Secrets of the Glasgow Haskell Compiler inliner. Journal of Functional Programming 12 (4–5), 393–434.

Rodriguez, A., Jeuring, J., Jansson, P., Gerdes, A., Kiselyov, O., Oliveira, B. C. d. S., 2008. Comparing libraries for generic programming in Haskell. Tech. Rep. UU-CS-2008-010, Utrecht University.

Rodriguez Yakushev, A., 2009. Towards getting generic programming ready for prime time. Ph.D. thesis, Utrecht University.

Santos, A., 1995. Compilation by transformation in non-strict functional languages. Ph.D. thesis, University of Glasgow.

Sculthorpe, N., Farmer, A., Gill, A., 2013. The HERMIT in the tree: Mechanizing program transformations in the GHC core language. In: Implementation and Application of Functional Languages.

Sculthorpe, N., Frisby, N., Gill, A., 2014. The Kansas University Rewrite Engine: A Haskell-embedded strategic programming language with custom closed universes. Journal of Functional Programming.

Turchin, V. F., Feb. 1979. A supercompiler system based on the language REFAL. ACM SIGPLAN Notices 14 (2), 46–54.

Vytiniotis, D., Peyton Jones, S., Magalhães, J. P., 2012. Equality proofs and deferred type errors: a compiler pearl. In: Proceedings of the 17th ACM SIGPLAN international conference on Functional programming. ICFP '12. ACM, New York, NY, USA, pp. 341–352.