Optimizing SYB Traversals Is Easy!

Michael D. Adams\textsuperscript{a,b,*}, Andrew Farmer\textsuperscript{c}, José Pedro Magalhães\textsuperscript{d}

\textsuperscript{a}School of Computing, University of Utah
\textsuperscript{b}Department of Computer Science, University of Illinois at Urbana-Champaign
\textsuperscript{c}Information and Telecommunication Technology Center, University of Kansas
\textsuperscript{d}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 hand-written, 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 eliminates this cost. Essentially, it is a partial evaluation that takes advantage of domain-specific knowledge about the structure of SYB. It optimizes SYB traversals to be as fast as handwritten, non-generic code, and benchmarks show that this optimization improves the speed of SYB traversals 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) \cite{Lammel} 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 \cite{Wrench}. It is easy to use and has strong support from the Glasgow Haskell Compiler (GHC) \cite{GHC}.

While SYB allows the easy and concise expression of traversals that otherwise require large amounts of handwritten code, it has a serious drawback, namely, poor runtime performance. Our own benchmarks show it to be an order of magnitude slower than handwritten, non-generic code, and this fact is
documented many times in the literature (Rodriguez Yakushev, 2009; Brown and Sampson, 2009; Chakravarty et al., 2009; Magalhães et al., 2010; Adams and DuBuisson, 2012; Sculthorpe et al., 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 traversals exhibit a structure that we can take advantage of in our optimizations. This paper presents a domain-specific optimization that transforms SYB traversals 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 and simplified form of supercompilation (Turchin, 1979, 1986) or partial evaluation (Jones et al., 1993) that uses type information to determine whether an expression should be computed statically at compile time or dynamically at runtime. It depends on having appropriate inlining and type information available and being able to monomorphise the traversal. 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 traversals 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 revised 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, \texttt{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 \texttt{mangle}. With SYB, however, we can use the \texttt{everywhere} and \texttt{mkT} functions to write this traversal simply as \texttt{everywhere (mkT mangle)}.

SYB defines many traversals in addition to \texttt{everywhere}, and the optimization presented in this paper handles these, but for the sake of simplicity our examples focus on the \texttt{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.

\begin{verbatim}
inc :: Int -> Int
inc n = n + 1

incrementSYB :: [Int] -> [Int]
incrementSYB x = everywhere (mkT inc) x
\end{verbatim}

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

We now turn to how \texttt{mkT} and \texttt{everywhere} work before considering why this SYB traversal is slower than an equivalent handwritten traversal.

2.1. Transformations

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

\begin{verbatim}
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
\end{verbatim}
2.2 Traversals

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.

```haskell
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 recur past the first layer of children, but by calling it with `everywhere f` as an argument, the `everywhere` function recurs to all the descendants of `x` in a bottom-up fashion.

2.3. Mapping subterms

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

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

The function `gmapT` is a method of the `Data` class and has a default implementation in terms of the SYB primitive `gfold1`, which has the following type.

```haskell
gfold1 :: (Data a)
    -> (∀d b. Data d => c (d -> b) -> d -> c b)
    -> (∀g. g -> c g) -> a -> c a
```

This type is very general. Informally, the first argument of `gfold1` is a function to chain together multiple arguments to a constructor. It is somewhat similar to the `Applicative` operation `<>`. The second argument is a function that produces a result when applied directly to a constructor. The third argument is a value to consume. Finally, `gfold1` returns a transformed value of type `c a` where the type constructor `c` can be used to change the return type into something other than `a`.

Since `gfold1` is a method of the `Data` class, its implementation is different for every type. However, for any constructor `C` with `n` arguments, `gfold1` should obey the following equality.
2.4 Why SYB is slow

\[ \text{gfoldl} \ k \ z \ (C \ x_1 \ldots x_n) = z \ C \ 'k' \ x_1 \ldots 'k' \ x_n \]

As we can see, the third argument, \( k \), chains the constructor and its arguments together. The second argument, \( z \), is applied to the constructor itself, while the third argument is the value over which the \text{gfoldl} method traverses. As an example, this pattern can be seen in the following class instance for lists.

\[
\begin{align*}
\text{instance Data a => Data [a] where} \\
\text{gfoldl} \ k \ z \ [] &= z \ [] \\
\text{gfoldl} \ k \ z \ (x:xs) &= z \ (::) 'k' \ x 'k' \ xs
\end{align*}
\]

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

\[
\begin{align*}
\text{gmapT} : : (\forall b. \text{Data } b \Rightarrow b \Rightarrow b) \\
&\Rightarrow (\forall a. \text{Data } a \Rightarrow a \Rightarrow a) \\
\text{gmapT} \ f \ x &= \text{unID} \ (\text{gfoldl} \ k \ \text{ID} \ x) \text{ where} \\
&k \ (\text{ID } c) \ y = \text{ID} \ (c \ (f \ y))
\end{align*}
\]

Since \text{gmapT} does not need to take advantage of the type changing ability provided by the \( c \) type parameter to \text{gfoldl}, it instantiates \( c \) to the trivial type \text{ID}. Aside from wrapping and unwrapping \text{ID}, \text{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 \text{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. Though these papers report varying performance overheads due to differing tests, benchmarking techniques, and compiler and SYB versions, all of these papers consistently 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 \texttt{incrementSYB} function. When applied to a value of type \texttt{[Int]} such as \texttt{[0,1]}, it recurs down the structure while applying \texttt{mkT} \texttt{inc} to every subterm. In this case, there are five subterms. Three of them are the lists \texttt{[0,1]}, \texttt{[1]} and \texttt{[]} . The remaining two are the \texttt{Int} values 0 and 1. For each subterm, \texttt{mkT} attempts to cast \texttt{inc} to have a type that is applicable to that subterm. On the lists, it fails to do so, and thus \texttt{mkT} returns them unchanged. On the \texttt{Int} values, however, the cast succeeds, and thus \texttt{mkT} applies \texttt{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 traversals. Since SYB relies heavily on runtime type comparison, the type specializer cannot guide the optimization as it does in the work of Magalhães \cite{Magalhaes2013}. Instead, in order to find out if \texttt{inc} can be applied to a term, we must inline \texttt{mkT}, \texttt{cast}, and the \texttt{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, \texttt{mkT inc} reduces to either \texttt{inc} or \texttt{id} depending on whether the types match. However, the GHC inliner \cite{PeytonJones2002}, 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 \cite{Magalhaes2010}. Furthermore, because \texttt{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 code is not clear.

3. Optimizing SYB traversals

In order to gain an intuition for optimizing SYB traversals, we now consider the \texttt{incrementSYB} 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.

\begin{verbatim}
incrementHand :: [Int] -> [Int]
incrementHand [] = []
incrementHand (x : xs) = inc x : incrementHand xs
\end{verbatim}

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

\begin{verbatim}
incrementSYB :: [Int] -> [Int]
incrementSYB = \lambda x \rightarrow everywhere e [Int] $dData x
\end{verbatim}

For presentation purposes, we abbreviate as \textit{e} the following expression, which occurs multiple times when optimizing the code.

\begin{verbatim}
\lambda b0 \$dData0 \rightarrow mkT Int b0 ($p1Data b0 \$dData0 \$fTypeableInt inc
\end{verbatim}

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 \texttt{$dData} and \texttt{$fTypeableInt} variables are \texttt{Data} and \texttt{Typeable} dictionaries for \texttt{[Int]} and \texttt{Int}, respectively, that were previously implicit. They are bound at the top-level and are automatically generated by the compiler when those class instances are declared. The \texttt{$p1Data b0 \$dData0} expression uses the automatically generated top-level function \texttt{$p1Data} to project \texttt{$dData0} from being a dictionary for \texttt{Data} to a dictionary for the superclass \texttt{Typeable}. We will see more such expressions as we proceed.

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

\begin{verbatim}
incrementSYB :: [Int] -> [Int]
incrementSYB = \lambda x \rightarrow
  mkT Int [Int] ($p1Data [Int] $dData) $fTypeableInt inc
  (gmapT [Int] $dData
   (\lambda b1 $dData1 \rightarrow everywhere e b1 $dData1)
    x)
\end{verbatim}

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

\begin{verbatim}
incrementSYB :: [Int] -> [Int]
incrementSYB =
  let $dTypeable4 = ...
    $dTypeable5 = ...
  in \lambda x \rightarrow
    (case cast (Int -> Int) ([Int] -> [Int])
      ...
This code attempts to cast \( \text{inc} \) from type \( \text{Int} \rightarrow \text{Int} \) to type \( \text{[Int]} \rightarrow \text{[Int]} \) by using the \texttt{cast} function. Inlining \texttt{cast} exposes calls to \texttt{typeOf} that we can symbolically evaluate. After several more simplification steps, this call to \texttt{cast} reduces to \texttt{Nothing}, and in turn the \texttt{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.

\[
\text{incrementSYB} :: \text{[Int]} \rightarrow \text{[Int]}
\]
\[
\text{incrementSYB} = \lambda \ x \rightarrow
\gmapT \ [\text{Int}] \ $dData
\quad (\lambda \ b1 \ $dData1 \rightarrow \text{everywhere} \ e \ b1 \ $dData1)
\ x
\]

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

Similar to before with \texttt{mkT}, we choose not to inline \texttt{everywhere} as it is not fully applied. Instead we inline \texttt{gmapT} and get the following code.

\[
\text{incrementSYB} :: \text{[Int]} \rightarrow \text{[Int]}
\]
\[
\text{incrementSYB} = \lambda \ x \rightarrow
\quad \text{case} \ x \ of
\quad \quad [] \rightarrow [] \text{ Int}
\quad \quad (:) \ x0 \ xs0 \rightarrow (:) \text{ Int} \ (\text{everywhere} \ e \ \text{Int} \ $fDataInt \ x0)
\quad \quad (\text{everywhere} \ e \ [\text{Int}] \ $dData \ xs0)
\]

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

The code now contains two calls to \texttt{everywhere} that are inside the \texttt{(:) branch of the \texttt{case} expression. One is on the head of the list and is at the type \texttt{Int}. The other is on the tail of the list and is at the type \texttt{[Int]}. We can inline the first of these which results in calls to \texttt{mkT} and \texttt{gmapT} just as before. This time, however, they are over the \texttt{Int} type. Thus, not only does the \texttt{cast in mkT} succeed and the \texttt{mkT} reduce to \texttt{inc}, but the call to \texttt{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) (incrementSYB 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 incrementSYB into incrementHand. 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 incrementSYB started with, and continuing to inline will thus lead us in a loop. Instead, we can take advantage of the fact that incrementSYB equals this expression and replace it with a reference to incrementSYB. Once we perform that replacement, we get the following code, which is identical to that of incrementHand.

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

4. A more principled attempt

The transformation in Section 3 is achieved by a simple combination of 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 type of code, namely SYB traversals, 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.

\[ e, u := x \]

| l | Variables |
| \( \Lambda a : \kappa. e \) | Literals |
| \( \lambda x : \sigma. e \) | Type abstraction and application |
| \( K \) | Term abstraction and application |
| \( \text{let} \ x : \tau = \hat{e} \text{ in } u \) | Constructors and \text{case} matching |
| \( e \triangleright \gamma \) | Local variable binding |
| \( e \downarrow \gamma \) | Casts |
| \( e \downarrow \gamma \) | Coercions as expressions |
| \( \rho := K \ x : \tau \) | Patterns |
| \( \tau := a | \forall a : \kappa. \tau | \tau_1 \tau_2 | \ldots \) | Types |
| \( \kappa := \star | \# | \kappa \rightarrow \kappa \) | Kinds |
| \( \gamma := \text{sym} \gamma \) | Symmetry rule for coercions |
| \( \text{nth} 1 \gamma \) | Arg part of function coercion |
| \( \text{nth} 2 \gamma \) | Result part of function coercion |
| \( \gamma @ \tau \) | Type application for coercions |
| \ldots \)

Figure 1: Syntax of System \( F_C \) (Excerpt)

\[ \begin{align*}
\text{BETA} & \quad (\lambda x : \tau. e_1) e_2 \quad \leadsto e_1 [e_2 / x] \\
\text{TyBeta} & \quad (\Lambda a : \kappa. e) \tau \quad \leadsto e [\tau / a] \\
\text{CASEBeta} & \quad \text{case } K \ e_1 \ldots \ K \ x_i : \tau_i \rightarrow e_j \ldots \ \leadsto e_j [e_i / x_i] \\
\text{Push} & \quad (e \triangleright \gamma) e_2 \quad \leadsto (e_1 (e_2 \triangleright \text{sym} (\text{nth} 1 \gamma))) \triangleright (\text{nth} 2 \gamma) \\
\text{TyPush} & \quad (e \triangleright \gamma) \tau \quad \leadsto (e \tau) \triangleright (\gamma @ \tau)
\end{align*} \]

Figure 2: Reductions of System \( F_C \) (Excerpt)
\( e : \tau \)  
Expression typing

\( \gamma : \tau_1 \sim \tau_2 \)  
Coercion typing

\( e \hookrightarrow e' \)
System \( F_C \) evaluation step (See Figure 2)

\( e \Rightarrow e' \)
Optimization step (See Figures 3, 7 and 8)

\( e \leftrightarrow e' \)
Symmetric cast elimination (See Figure 9)

\( e \overset{\sim}{\rightarrow} e' \)
Force step (See Figures 6 and 10)

\( e \overset{\sim}{\Rightarrow} e' \)
Deep force step (See Figure 10)

**Und \( \tau \)**  
Undesirable type

**ElimUnd \( e \)**  
Elimination expression (See Figure 4)

**Memo \( e \)**  
Memoizable expression (See Figure 7)

---

1. Replace any expression with a memoization that it matches as discussed in Section 4.4.

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 4.7.
   
   (a) Memoize an expression having an undesirable type using the rules from Figure 7.
   
   (b) Eliminate an expression having an undesirable type using the rules from Figure 4.

---

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

With the benchmarks in Section 5 we show that this algorithm successfully optimizes typical SYB traversals 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 traversals. However, the transforma-
tions performed in Section 3 allowed us to eliminate expressions with those types
from the code for incrementSYB. 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. These in turn contain Fingerprint objects that contain hashes
for efficiently comparing the TypeRep and TyCon values. Finally, the default
implementations of several of the methods in the Data class use the newtype
wrappers ID, CONST, Qi, Qr, and Mp that interfere with the optimization process
and should also be eliminated.

In Section 3 we were able to eliminate expressions that have these unde-
sirable 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
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 the reduction rules for these elimination forms can be applied. 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 ELIMUNAPP, ELIMUNCASE, and ELIMUNCAST rules, and the rules
for forcing a step of evaluation are specified in Figure 6. These rules use the
Und \( \tau \) judgment, which holds if and only if the type \( \tau \) syntactically contains an
occurrence of an undesirable type and is defined in Figure 5. Note that we treat
a type application, \( \tau_1 \tau_2 \), as undesirable if either \( \tau_1 \) or \( \tau_2 \) is undesirable. The
former is due to undesirable types like ID that take another type as parameter,
and the latter is due to internal SYB operations that construct intermediate
structures such as lists containing undesirable types. Finally, we use typing
judgments for expressions, \( e : \tau \), and coercions, \( \gamma : \tau_1 \sim \tau_2 \). These judgments
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 `incrementSYB`. In general, however, we cannot rely on such a binding
4.3 Simplification

\[ \tau \in \{ \text{Data, Typeable, TypeRep, TyCon, Fingerprint, ID, CONST, Qi, Qr, Mp} \} \]

\[ \text{Und } \tau \]

\[ \text{Und } \tau_1 \quad \text{Und } \tau_2 \quad \text{Und } \tau \]

\[ \text{Und } \tau \]

\[ \text{Und } (\tau_1 \tau_2) \quad \text{Und } (\tau_1 \rightarrow \tau_2) \quad \text{Und } (\forall x : \kappa . \tau) \]

Figure 5: Undesirable Types

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 7. 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.

If \( e \) ever occurs again, `MEMOREPLACE` fires and replaces 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.

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.

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 7. 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
4.3 Simplification

\textbf{ForceBeta}

\((\lambda x : \tau. e_1) \ e_2 \quad \leadsto \quad \text{let} \ x : \tau = e_2 \ \text{in} \ e_1\)

\textbf{ForceTyBeta}

\((\Lambda a : \kappa. e) \ \tau \quad \leadsto \quad \text{let} \ a : \kappa = \tau \ \text{in} \ e\)

\textbf{ForceCaseBeta}

\text{case} \ K \ e'_i \ \text{of} \ ... \ K \ x_i : \tau_i \rightarrow e_j \ ...
\quad \leadsto \quad \text{let} \ x_i : \tau_i = e'_i \ \text{in} \ e_j

\textbf{ForcePush}

\((e_1 \triangleright \gamma) \ e_2 \quad \leadsto \quad (e_1 \ (e_2 \triangleright \text{sym} \ (\text{nth} \ 1 \ \gamma)) \triangleright \ (\text{nth} \ 2 \ \gamma))

\textbf{ForceTyPush}

\((e \triangleright \gamma) \ \tau \quad \leadsto \quad (e \ \tau) \triangleright \ (\gamma \triangleright \tau)\)

\textbf{ForceVar}

\(x \quad \leadsto \quad e \quad \text{if} \ e \ \text{is the inlining of} \ x\)

\textbf{ForceLetFloatApp}

\((\text{let} \ x : \tau = e'_i \ \text{in} \ e_0) \ u \quad \leadsto \quad \text{let} \ x : \tau = e'_i \ \text{in} \ e_0 \ u\)

\textbf{ForceLetFloatScr}

\text{case} \ (\text{let} \ x : \tau = \hat{u} \ \text{in} \ e_0) \ \text{of} \ ... \ \text{case} \ e_0 \ \text{of} \ ...
\quad \leadsto \quad \text{let} \ x : \tau = \hat{u} \ \text{in} \ (\text{case} \ e_0 \ \text{of} \ ...)

\textbf{ForceAppFun}

\(e_1 \ e_2 \quad \leadsto \quad e'_1 \ e_2 \quad \text{if} \ e_1 \leadsto e'_1\)

\textbf{ForceAppTyFun}

\(e_1 \ \tau \quad \leadsto \quad e'_1 \ \tau \quad \text{if} \ e_1 \leadsto e'_1\)

\textbf{ForceScr}

\text{case} \ e_0 \ \text{of} \ p_i \rightarrow e'_i
\quad \leadsto \quad \text{case} \ e'_0 \ \text{of} \ p_i \rightarrow e'_i \quad \text{if} \ e_0 \leadsto e'_0\)

\textbf{ForceLetBody}

\text{let} \ x_i : \tau_i = u_i \ \text{in} \ e
\quad \leadsto \quad \text{let} \ x_i : \tau_i = u_i \ \text{in} \ e' \quad \text{if} \ e \leadsto e'\)

\textbf{ForceCast}

\(e \triangleright \gamma \quad \leadsto \quad e' \triangleright \gamma \quad \text{if} \ e \leadsto e'\)

Figure 6: Forcing Rules
were chosen by examining the sorts of code generated when optimizing SYB traversals 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 small 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 \texttt{newtype} definitions and some class dictionaries makes use of casts. For example, a call to the \texttt{typeOf} method of the \texttt{Typeable} class is implemented as a cast. In addition, many of the SYB functions use \texttt{newtype} definitions to define higher level operations in terms of lower level operations. For example, the default implementation of \texttt{gmapT} is in terms of \texttt{gfoldl} with \texttt{ID} for the \texttt{c} type parameter. Since \texttt{ID} is a \texttt{newtype}, calls to the \texttt{ID} constructor and \texttt{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 \texttt{FORCETyBeta} or shuffled around by the \texttt{FORCEPush} or \texttt{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 \texttt{case} contains a reflexive cast wrapped around a constructor. Until we eliminate the cast, we cannot use the \texttt{FORCECaseBeta} rule even though the constructor involved is already manifest.

Reflexive casts from a type to itself are directly eliminated with the \texttt{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
4.4 Primitives

\begin{align*}
\text{CastRefl} & \quad e \triangleright \gamma \quad \rightarrow e \quad \text{if } \gamma : \tau \sim \tau \\
\text{CastSym} & \quad e \triangleright \gamma \quad \rightarrow e' \quad \text{if } e \overset{\gamma}{\rightarrow} e' \\
\text{DeadLet} & \quad \text{let } x : \tau = u \text{ in } e \quad \rightarrow e \quad \text{if } x \not\in \text{fv}(e) \text{ and } x \text{ is not a memoization} \\
\text{SubstStar} & \quad \text{let } x : \star = \tau \text{ in } e \quad \rightarrow e[\tau/x] \\
\text{SubstHash} & \quad \text{let } x : \# = \tau \text{ in } e \quad \rightarrow e[\tau/x] \\
\text{SubstVar} & \quad \text{let } x : \tau = x' \text{ in } e \quad \rightarrow e[x'/x] \\
\text{SubstLit} & \quad \text{let } x : \tau = l \text{ in } e \quad \rightarrow e[l/x] \\
\text{SubstDFun} & \quad \text{let } x : \tau = v\overrightarrow{u} \text{ in } e \quad \rightarrow e[v\overrightarrow{u}/x] \quad \text{if } v \text{ is a dictionary constructor}
\end{align*}

Figure 8: Simplifications

where \( \gamma_1 : \tau_1 \sim \tau_2, \gamma_2 : \tau_2 \sim \tau_1, \) and \( \gamma_3 : \tau_2 \sim \tau_1. \)

\[(\text{case } x \text{ of } \{ C_1 \rightarrow e_1 \triangleright \gamma_2; C_2 \rightarrow e_2 \triangleright \gamma_3 \}) \triangleright \gamma_1\]

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

\[\text{case } x \text{ of } \{ C_1 \rightarrow e_1; C_2 \rightarrow e_2 \}\]

4.3.2. Let elimination

We also eliminate let bindings that are either trivial, dead, or bind a type as they may interfere with our ability to apply the core optimization rules. These are implemented by the remaining rules in Figure 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. These objects contain unique hashes in Fingerprint objects. These hashes are recursively computed from the hashes of the contents of the TypeRep objects by the fingerprint function. Equality over TypeRep objects is then implemented by comparing
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 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 \texttt{mkT}. When optimizing SYB traversals, this often results in intermediate residual code with a structure similar to the following.

\begin{verbatim}
case typeOf t1 == typeOf t2 of
  True -> ...
  False -> ...
\end{verbatim}

The equality operator in this code is over the undesirable type \texttt{TypeRep}, so the optimization will reduce it to either \texttt{True} or \texttt{False}. After that, the scrutinee no
4.5 Optional optimizations

Figure 10: Rules for Primitives
longer contains an expression with an undesirable type, so the core optimization does not then simplify the `case` expression even though it has a known constructor in its scrutinee. In most cases this is not a problem as the code to be optimized under each branch of the `case` expression tends to be small, and we can simply rely on downstream optimizations to simplify the `case` expression. However, when these branches are large, they can represent a significant amount of extra work to be done by the optimization. It would be better to detect the dead branch and skip the extra work in that branch. To do this we apply the rewrite in `ForceCaseBeta` whenever possible. This rewrite never makes the code worse or worsens the optimization result. Note that our use of this rewrite differs from the usual use of the rules in Figure 6 since we apply it at any position in the expression regardless of whether it eliminates an expression with an undesirable type.

4.5.2. Memoization floating

Duplicate memoizations of the same expression may arise if the first memoization is not in scope at the other occurrences of the same expression. For example, when traversing an abstract syntax tree, memoizations of the traversal at the identifier type may occur inside both the part of the code for \(\lambda\)-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 \(\lambda\)-expression floats outwards, the traversal for the identifier in a `let` expression can use the existing memoization instead of creating a new one. We also consolidate memoization bindings into a common recursive `let` 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 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, declar-
ative 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 facili-
ties, 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 bench-
marked 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 Ints 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 struc-
ture. It is implemented in SYB using everywhere and mkT. This traversal
is performed on three data types. The first is a binary tree of integers. The
second is a logic formula. The third is an AST type from the haskell-src
module involving over 30 types and 100 different constructors. For the
binary tree, all integers are incremented. For the other two types, all
characters are replaced with the character ‘y’.

RenumberInt Taken from Adams and DuBuisson (2012), the RenumberInt
benchmark replaces each integer in a structure with a new, unique integer
that is drawn from a state monad. This traversal is also performed on
both binary tree and logic formula data types. It is implemented in SYB
using everywhereM and mkM.
The `Map` benchmarks applied the generic traversal in a context in which all types are known. The other benchmarks first defined their traversals polymorphically in one module and used them on concrete types in another module. For example, `RmWeights` defined a function `rmWeights` with type `Data a => a -> a` in one module and used that on the weighted binary tree type in another module.

To ensure that these benchmarks are representative of real world uses of SYB, we surveyed the Hackage repository and found 246 uses of SYB traversal functions in third-party packages. After examining each of these by hand, we found that the vast majority (93%) of calls to traversals were simple and passed the result of a single call to `mkT`, `mkQ`, or `mkM` to the traversal to define the transformation to be performed by the traversal. The remainder were only minor variations of this such as `extT` on top of a `mkT`.

We also examined whether the traversals were over concrete types or abstract, polymorphic types. We found that 58% were used in contexts where all types were known. These are direct candidates for our optimization. A further 29% were used to define polymorphic functions that were then used in contexts where all types were known. These uses would also be candidates for our optimization. Only 13% did not fall into these two categories.

### 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 Magaes et al. (2010) was used to run each program 10 times and take the average running time. We compiled the benchmarks with GHC 7.8.4 using the `--02` 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. Similarly, the Hand implementation of `RenumberInt` in GPBench did not descend into strings, whereas the SYB implementation did. We modified the Hand implementation to match the SYB traversal. This slowed down the Hand implementation by 36%. These changes ensure that our benchmarks measure the overhead due to SYB instead of algorithmic differences.

### 5.3. Performance results

Figure [1] 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 hand-written code.
5.3 Performance results

For all of the benchmarks except \texttt{RmWeights}, the optimization completely eliminates the runtime costs associated with SYB. A manual inspection of the generated \texttt{Core} confirms that the optimization does indeed eliminate all runtime type checks and dictionary dispatches in the SYB traversals and that the resulting code is equivalent to the handwritten code.

When initially running these benchmarks, the SYB/Hermit versions of \texttt{MapTree} and \texttt{MapLogic} actually ran faster than the Hand versions by about 20%. Analysis of the resulting \texttt{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 \cite{Santos1995} improved its performance to match that of the SYB/Hermit version.

5.3.1. The performance of \texttt{RmWeights}

On the \texttt{RmWeights} benchmark, SYB/Hermit fails to achieve parity with the Hand version. This contrasts with a previous version of this paper\cite{Adams2014} in which \texttt{RmWeights} fully optimized. Inspecting the \texttt{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.)

\begin{verbatim}
memo_everywhere :: WTree Int Int -> WTree Int Int
memo_everywhere = \lambda x ->
case memo_gfoldl x of
\end{verbatim}
5.3 Performance results

\[
\begin{align*}
\text{WithWeight } t w & \to t \\
\text{wild } & \to \text{wild} \\
\text{memo\_gfoldl} :: \text{WTree } \text{Int } \text{Int} & \to \text{WTree } \text{Int } \text{Int} \\
\text{memo\_gfoldl} & = \lambda \text{ds} \to \\
\text{case } \text{ds} \text{ of} \\
\text{Leaf } a1 & \to \text{Leaf } \text{Int } \text{Int} \ a1 \\
\text{Fork } a1 \ a2 & \to \\
\text{Fork } \text{Int } \text{Int} & (\text{memo\_everywhere } a1) \ (\text{memo\_everywhere } a2) \\
\text{WithWeight } a1 \ a2 & \to \\
\text{WithWeight } \text{Int } \text{Int} & (\text{memo\_everywhere } a1) \ a2
\end{align*}
\]

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.

5.3.2. The performance of `RenumberInt`

In the previous version of this paper, `RenumberIntLogic` 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 RenumberInt\textsubscript{Tree} and RenumberInt\textsubscript{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 RenumberInt\textsubscript{Tree} and RenumberInt\textsubscript{Logic} in Figure 11 are for the strict State monad.

5.3.3. The performance of SelectInt

Finally, we also benchmarked the quadratic algorithm for SelectInt with Hand, SYB, and SYB/Hermit. The SYB and SYB/Hermit versions ran 4.4 and 2.9 times slower than Hand, respectively. In order to see why the optimization did not fully eliminate the overhead in SYB/Hermit, note that the quadratic algorithm calls everything (++) (mkQ [] (\x -> [x])) on the object being traversed, and everything is implemented as follows.

\[
\text{everything} :: (r \to r \to r) \\
\to (\forall b. \text{Data } b \Rightarrow b \to r) \\
\to (\forall a. \text{Data } a \Rightarrow a \to r) \\
\text{everything } k f x = \text{foldl} k (f x) (\text{gmapQ} (\text{everything } k f) x)
\]

This uses the \text{gmapQ} function, which has the following type.

\[
\text{gmapQ} :: (\forall b. \text{Data } b \Rightarrow b \to r) \\
\to (\forall a. \text{Data } a \Rightarrow a \to [r])
\]

The \text{gmapQ} function is a method of the \text{Data} class and obeys the following equality.

\[
\text{gmapQ } f (C \ x_1 \ldots x_n) = [f \ x_1, \ldots, f \ x_n]
\]

After our optimization runs, the code contains two mutually recursive functions similar to what we saw with \text{RmWeights} in Section 5.3.1. One of these functions implements the \text{gmapQ} part of the traversal and returns a list of results for each child. The other implements the \text{foldl} part of the traversal and concatenates those results together.

Constructing a list only to immediately eliminate it with a \text{foldl} is obviously inefficient. Unfortunately, GHC chooses the function implementing the \text{gmapQ} part of the traversal as a loop breaker. This prevents it from being inlined into the function implementing the \text{foldl} part. Thus GHC never notices the use of \text{foldl} on a freshly constructed list and does not sufficiently optimize that part of the code. As with \text{RmWeights}, forcing this inlining avoids this problem and improves the performance to match that of Hand.

We can also avoid these issues by changing the definition of \text{everything} to use \text{gmapQl} instead of \text{gmapQ}. The \text{gmapQl} function skips constructing an intermediate result list and directly performs a left fold. It obeys the following equality.

\[
\text{gmapQl } k z f (C \ x_1 \ldots x_n) = z 'k' f x_1 \ldots 'k' f x_n
\]
The `everything` function can then be implemented as the following, which is semantically equivalent to the original `everything`.

\[
\text{everything } k \ f \ x = \text{gmapQI } k \ (f \ x) \ (\text{everything } k \ f) \ x
\]

With this implementation, the SYB version still ran 4.0 times slower than Hand, but the SYB/Hermit version fully optimized to match the performance of Hand.

6. Limitations and future work

While the algorithm described in Section 4 is effective for most instances of SYB traversals, 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 this optimization is driven by the presence of expressions with undesirable types, code that does not contain expressions with those types should not be adversely effected. However, the optimization is still domain-specific and may not be appropriate for all code. The limitations of our current implementation may cause it to fail on certain types of SYB 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.

\[
\text{mapSYB} \ : \ (\text{Data } a ) \Rightarrow (a \rightarrow a) \rightarrow [a] \rightarrow [a]
\]

\[
\text{mapSYB } f \ x = \text{everywhere } (\text{mkT } f) \ x
\]

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 \texttt{mapSYB} but also to the sub-lists of those elements.
Until we know a, it is impossible to know how to traverse those elements.

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

\begin{verbatim}
incrementSYB/Int :: [Int] -> [Int]
incrementSYB/Int = mapSYB inc
\end{verbatim}

Because this definition completely determines the type of a in \texttt{mapSYB} and thus
calls \texttt{mapSYB} with a concrete \texttt{Data} dictionary for a, the optimization will success-
fully complete on \texttt{incrementSYB/Int}, even though it would fail on \texttt{mapSYB}.

Finally, note that specialized versions of \texttt{mapSYB} that are successfully opti-
mized by our optimization can be explicitly generated by specifying their types
as in the following.

\begin{verbatim}
mapSYB/Int :: (Int -> Int) -> [Int] -> [Int]
mapSYB/Int = mapSYB
\end{verbatim}

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 \texttt{TypeRep}
or \texttt{TyCon}, but less obvious examples of this include types like the following from
\cite{Hinze2006}.

\begin{verbatim}
data Spine b
  = Unit b
  | ∀ a. (Data a) => App (Spine (a -> b)) a
\end{verbatim}

Here the existential\footnote{GHC uses the $\forall$ keyword for both existential and universal types. The distinction between
the two is where the keyword is placed.} type a is qualified by the \texttt{Data} class and thus the \texttt{App}
constructor contains a dictionary for the \texttt{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 \texttt{TypeRep} and \texttt{TyCon} objects. This may result in occurrences of unde-
sirable 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. The majority of types in Haskell programs are regular, but non-regular, polymorphically recursive types do occur. Consider, for example, the following polymorphically recursive, non-regular type.

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

If we attempt to traverse over the type \texttt{T Int}, then the traversal will initially be memoized at \texttt{T Int}. Since at this type the argument to the \texttt{Double} constructor is of type \texttt{T (Int, Int)}, the traversal will also have to be memoized at type \texttt{T (Int, Int)}. In turn, at that type, the argument to the \texttt{Double} constructor has type \texttt{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 \texttt{T Int} type effectively contains an infinite number of types.

Note that this pattern frequently occurs in GADTs, but the difficulty there is not the GADT itself. It is the infinite number of types that decedent terms could have. GADTs with only a finite number of possible types for decedents do not pose this problem.

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 \texttt{Int} inside an object of type \texttt{T Int}. The generic code for this is the following.

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

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

```haskell
incrementT :: T Int -> T Int
incrementT x = go inc x where
    go :: (a -> a) -> T a -> T a
```

```haskell
      = Base a
```
6.4 Polymorphic recursion in terms

\[ \text{go } f \ (\text{Base } x) = f \ x \]
\[ \text{go } f \ (\text{Double } t) = \text{Double } (\text{go } (f' \ f) \ t) \]
\[ f' :: (a \to a) \to (a, a) \to (a, a) \]
\[ f' \ f \ (x1, x2) = (f \ x1, f \ x2) \]

Since the optimization presented in this paper preserves the structure of the generic traversal and incrementT does not follow that structure, it is unsurprising that our optimization fails on such a traversal. However, note that the \( f \) argument to \( \text{go} \) serves essentially the same role as the \texttt{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 \texttt{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.

\[ \text{poly} :: (\forall b. \texttt{Data } b \Rightarrow b \to b) \]
\[ \to (\forall a. \texttt{Data } a \Rightarrow a \to a) \]
\[ \text{poly } f \ x = f \ (\text{gmapT } (\text{poly } (f \ '\texttt{extT'} \ g)) \ x) \]
\[ \text{where } g = \ldots \]

Note how the \( f \) argument to the traversal is extended each time through the traversal. As a result, the previously memoized instances of \texttt{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. \cite{AdamsDuBoisson2012} 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.

```haskell
memoChar c    = c
memoString [] = []
memoString (c : cs) = memoChar c : memoString cs
```

Here `memoString` is equivalent to the identity function and can thus be more
efficiently implemented by not doing the traversal and simply returning its ar-
gument. 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 partic-
ular 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 `incrementSYB` 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 traversals.

The situation is not a total loss, however. Under appropriate conditions
the GHC specializer will specialize some parts of `incrementSYB` over the `[Int]`
type and produce the following code.

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

```haskell
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 `incrementSYB` results in the following.

```haskell
incrementSYB :: [Int] -> [Int]
incrementSYB x = go x 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.

```haskell
incrementSYB :: [Int] -> [Int]
incrementSYB x = go[Int] x 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[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[Int]` to the following.
Unfortunately, in our experiments with GHC the optimization process often simplified \texttt{mkT} and the contained \texttt{cast} but did not do the final step of removing the comparison over \texttt{TypeRep} objects. This seems to be due to the simplifier not knowing how to symbolically evaluate the \texttt{fpFingerprints} 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 \texttt{gmapT} inside \texttt{go[Int]}. It is also over the concrete type \texttt{[Int]} so the simplifier statically computes the dictionary dispatch and changes the code to use \texttt{gmapT[Int]}, the \texttt{gmapT} implementation in the \texttt{Data} instance for \texttt{[Int]}. This results in the following code.

\begin{verbatim}
  go[Int] :: [Int] -> [Int]
  go[Int] x = gmapT go x
\end{verbatim}

Unfortunately, the simplifier does not inline this invocation of \texttt{gmapT[Int]}. This is because there is a cycle between the dictionary for \texttt{Data} at the \texttt{[Int]} type and \texttt{gmapT[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[Int]} is marked. As a result, the simplifier does not inline \texttt{gmapT[Int]}. If we overlook this obstacle and force \texttt{gmapT[Int]} to inline, then we get the following code.

\begin{verbatim}
  go[Int] :: [Int] -> [Int]
  go[Int] [] = []
  go[Int] (x : xs) = go x : go xs
\end{verbatim}

This exposes two calls to \texttt{go}. One is on \texttt{\textbf{x}} and is over the \texttt{Int} type. The other is on \texttt{\textbf{xs}} and is over the \texttt{[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 \texttt{go} over the \texttt{[Int]} type to \texttt{\textbf{go[Int]}}.

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

\begin{verbatim}
  incrementSYB :: [[Int]] -> [[Int]]
  incrementSYB x = go x where
  go :: ∀c. Data c => c -> c
  go x = mkT inc (gmapT go x)
\end{verbatim}
This code contains a manifest call to \texttt{go} at the \texttt{[[Int]]} type, but there is no such manifest call to \texttt{go} at the \texttt{[Int]} type. This is because \texttt{go} is passed as a polymorphic argument to \texttt{gmapT}. This is exactly the sort of argument that the GHC specializer does not know how to handle. With \texttt{everywhere} we were able get around this by doing a static argument transformation, but \texttt{gmapT} is a class method so we cannot do the same. Thus when the GHC specializer runs, \texttt{go} is specialized only at the \texttt{[[Int]]} type. This results in the following code.

\begin{verbatim}
incrementSYB :: [[Int]] -> [[Int]]
incrementSYB x = go[[Int]] x 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)
\end{verbatim}

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

\begin{verbatim}
go[[Int]] :: [[Int]] -> [[Int]]
go[[Int]] [] = []
go[[Int]] (x : xs) = go x : go xs
\end{verbatim}

The call \texttt{go x s} is over the type \texttt{[[Int]]} for which we have a specialization so this is then turned into the following.

\begin{verbatim}
go[[Int]] :: [[Int]] -> [[Int]]
go[[Int]] [] = []
go[[Int]] (x : xs) = go x : go[[Int]] xs
\end{verbatim}

As before, we have the call \texttt{go x} over a type which does not have a specialization. This time, however, it is over the type \texttt{[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 traversal. However, note that after specialization and simplification, we now have a manifest call of \texttt{go} on the \texttt{[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[[Int]]} function. After another round of inlining and simplification this results in the following.

\begin{verbatim}
incrementSYB :: [[Int]] -> [[Int]]
incrementSYB x = go[[Int]] x 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
\end{verbatim}
This in turn has exposed a call to $\text{go}$ on the $\text{Int}$ type inside $\text{go[Int]}$. 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.

```haskell
g0[Int] :: [Int] -> [Int]
g0[Int] [] = []
g0[Int] (x : xs) = g0 x : g0[Int] xs
```

Each time we invoke the specializer and then the simplifier, we potentially uncover more types at which to specialize $\text{go}$. 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 traversals, a few modifications are sufficient to do so. First, we static-argument transform traversals like $\text{everywhere}$. 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 $\text{go}$ function resulting from the inlining of $\text{everywhere}$ is specialized to particular types. Third, we run the simplifier again with two modifications. We symbolically evaluate $\text{TypeRep}$, $\text{TyCon}$, and $\text{Fingerprint}$ calculations using rules such as those in Figure 10. We also inline the type-specific instances of $\text{gfoldl}$, $\text{gmapT}$, $\text{gmapQ}$, and $\text{gmapM}$ 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 $\text{go}$. The end result of all this is a fully optimized version of the code that contains no $\text{Data}$ or $\text{Typeable}$ class dispatches and no $\text{TypeRep}$, $\text{TyCon}$, or $\text{Fingerprint}$ computations.

Note that using the specializer in this way is not as general as the optimization described in Section 2. It relies on the code having a structure similar to $\text{everywhere}$ 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 $\text{gfoldl}$, $\text{gmapT}$, $\text{gmapQ}$, or $\text{gmapM}$
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 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 handwritten code. There has been a significant amount of work on designing more
efficient generic-programming systems (Mitchell and Runciman 2007; Brown
and Sampson 2009; Chakravarty et al. 2009; Augustsson 2011; Adams and
DuBuisson 2012), but there is little work on optimizing a pre-existing generic-
programming system as we do here. Magalhães (2013) shows how to optimize
the generic-deriving system by using standard compiler optimizations, but
notes that his techniques are not sufficient to optimize SYB traversals. Ali-
marine 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
specializes over only manifest constructors. As discussed in Section 7, dictionary
specialization is not sufficient to optimize SYB traversals, but using the lessons
and experience from our work we were able to find modifications of the GHC
specializer to effectively optimize SYB traversals.

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, 1986). 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 6. 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 traversals. 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 traversals, 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 traversals, 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 traversals. 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.

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, 1117569, and 1248464, the Rockwell Collins contract 4504813093, and the Engineering and Physical Sciences Research Council (EPSRC) grant EP/J010995/1.

References


38


