Template Your Boilerplate

Using Template Haskell for Efficient Generic Programming

Michael D. Adams Thomas M. DuBuisson
Portland State University

Abstract

Generic programming allows the concise expression of algorithms that would otherwise require large amounts of handwritten code. A number of such systems have been developed over the years, but a common drawback of these systems is poor runtime performance relative to handwritten, non-generic code. Generic-programming systems vary significantly in this regard, but few consistently match the performance of handwritten code. This poses a dilemma for developers. Generic-programming systems offer concision but cost performance. Handwritten code offers performance but costs concision.

This paper explores the use of Template Haskell to achieve the best of both worlds. It presents a generic-programming system for Haskell that provides both the concision of other generic-programming systems and the efficiency of handwritten code. Our system gives the programmer a high-level, generic-programming interface, but uses Template Haskell to generate efficient, non-generic code that outperforms existing generic-programming systems for Haskell.

This paper presents the results of benchmarking our system against both handwritten code and several other generic-programming systems. In these benchmarks, our system matches the performance of handwritten code while other systems average anywhere from two to twenty times slower.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.1.2 [*Programming Techniques*]: Automatic Programming

General Terms Algorithms, Design, Languages, Performance.

Keywords Generic programming, Scrap your Boilerplate, Template Haskell

1. Introduction

Generic programming provides a concise way to express many algorithms. In particular, many data-structure transformations and queries can be expressed without manually writing the uninteresting parts of data-structure traversal. For example, consider the task of collecting every variable in an abstract syntax tree (AST). Many sub-terms of various types must be traversed even though we are interested in only the parts dealing with variables. Other parts of the traversal follow a predictable pattern and are "boilerplate code" [Lämmel and Peyton Jones 2003]. Generic programming allow the programmer to focus on the interesting parts of the code and leave the boilerplate to the generic-programming system. This frees the developer from the drudgery of writing boilerplate code and makes it possible to concisely express these traversals.

Many generic-programming systems have focused on theoretically elegant constructions and increasing expressivity, but performance should also be a consideration lest the resulting system be too slow for practical use. Unfortunately, among generic-programming systems poor performance is a common problem.

We faced this performance problem when implementing the compiler for the Habit language [HASP Project 2010]. When choosing a generic-programming system, our goals were completely pragmatic: the system needed to perform well and be easy to use. Using generic programming made it easy to factor the Habit compiler into smaller passes. As a result, the code became easier to manage and debug. Each new pass, however, incurs overhead as the AST is traversed multiple times. Thus, any overhead introduced by a generic-programming system is paid multiple times and should be minimized. Ideally, passes written using generic programming should be just as fast as handwritten passes. This made existing generic-programming systems ill-suited to our needs.

We turned to Template Haskell [Sheard and Peyton Jones 2002] to solve these performance problems, and this paper documents the results of that work. Sheard and Peyton Jones allude to the possibility of using Template Haskell for generic programming, but working directly with Template Haskell can be a daunting task. The primitives introduced in this paper greatly simplify this by abstracting the user from the more complex parts of Template Haskell and thus allow the user to write traversals in a high-level style on par with other generic-programming systems.

In particular, we show that, with appropriate library primitives, generic programming in Template Haskell can be as concise as in other generic-programming systems while maintaining the performance of handwritten code. We have implemented these primitives in the *Template Your Boilerplate* (TYB) library, which is available at `http://hackage.haskell.org/package/TYB`.

Section 2 of this paper is a short review of the essentials of Template Haskell. Section 3 demonstrates the ease of use of TYB. Section 4 details how TYB is implemented. Section 5 demonstrates the performance of TYB. Section 6 reviews related work. Section 7 concludes.

In this paper we use *Scrap Your Boilerplate* (SYB) as a reference point when explaining some concepts. Basic knowledge of SYB will be helpful to the reader but is not necessary. We do not assume any prior knowledge of Template Haskell.

Haskell'12, September 13, 2012, Copenhagen, Denmark.

2. A Crash Course in Template Haskell

We now review the essential ideas of Template Haskell used in this paper. Readers already familiar with Template Haskell can safely skip this section.

Template Haskell is a metaprogramming system for Haskell and has been integrated as a standard part of GHC since version 6.0. As a metaprogramming system, Template Haskell allows parts of programs to be pragmatically generated at compile time instead of being directly written by the programmer. It consists of a library for representing the ASTs of Haskell code fragments as well as syntactic constructs for quotes and splices. This section describes these and shows examples of their use.

The AST types include identifiers (`Name`), expressions (`Exp`), and types (`Type`). In addition, Template Haskell includes a monadic type (`Q`) that acts as an interface between the compiler and the Template Haskell code that executes at compile time. It is used to query the environment as well as to generate fresh identifiers.

Template Haskell includes several quotation constructs as a convenience for defining AST objects. They act as literals and have values that are the AST of the Haskell code fragment written inside them. They make it easier to construct ASTs by avoiding the need to directly reference low-level AST constructors. There are multiple types of quotations but only the ones for `Name` ($'\cdot$), `Exp` ($[\![\cdot]\!]$), and `Type` ($[\![_t \cdot]\!]$) are used in this paper.

An example of these is `'map` which returns the fully qualified `Name` of the `map` function. Likewise, $[\![$ `λx → 1 + x` $]\!]$ returns a `Q Exp` that represents the AST of the expression `λx → 1 + x`. Finally, $[\![_t$ `∀ a. Int → a → Int` $]\!]$ returns a `Q Type` that represents the AST for the type `∀ a. Int → a → Int`. The `Q` monad in the return values of these quotations is used to generate fresh variable names for `x` and `a`.

The final syntactic construct is the splice. It is written `$(`e`)` and can appear anywhere either an expression or a type is expected. If a splice occurs where an expression is expected, then e must be of type `Q Exp`. If it occurs where a type is expected, then e must be of type `Q Type`. In either case, the AST fragment returned by e is inserted where the splice occurs.

If the splice is inside a surrounding quotation, then the AST returned by the quotation will contain the AST returned by e at the position of the splice. For example, the following defines `mkInc` to be a function that generates an AST for an increment function with an increment based on the value of `exp`.

```
mkInc :: Q Exp → Q Exp
mkInc exp = [|λx → x + $(exp)|]
```

If a splice is *not* inside a quotation, then it is a top-level splice, and e is executed at compile time instead of runtime. The compiler uses the AST returned by e in place of the splice. For example, one could use the `mkInc` function and a hypothetical `randomInt` function to randomly choose[1] an increment at compile time:

```
main = print (inc 3) >> print (inc 42) where
  inc = $(mkInc randomInt)
```

If `randomInt` returns an AST fragment for the literal `17` during compilation, then this code compiles as if `inc` is `λx → x + 17`.

The relationship between quotes and splices in Template Haskell is similar to the relationship between `quasiquote` and `unquote` in Scheme and Lisp [Bawden 1999] in that splices cancel out a surrounding quote. They differ in that top-level splices, do not have a surrounding quote and delimit parts of the program that are generated at compile time.

[1] Non-deterministic functions, such as `randomInt`, are possible due to the `IO` monad embedded in the `Q` monad used by Template Haskell.

3. TYB Examples

As an example of using TYB, consider the task of manipulating a real-world AST. For example, the AST from `Language.Haskell.Syntax` includes not just expressions but also declarations, statements, patterns and many other forms.

Suppose we wish to prefix every identifier (`HsName`) in a module (`HsModule`) with an underscore. In traversing an `HsModule`, over 30 different types with over 100 constructors are potentially involved. Given the number of types and constructors, few programmers would look forward to implementing such a thing by hand. Generic-programming systems like SYB make this easy. For example, in SYB, the `everywhere` function traverses a value of arbitrary type and applies a supplied transformation to every subterm. Thus, we can write:

```
prefixNames_syb x = everywhere (mkT f) x where
  f :: HsName → HsName
  f name = prefixName "_" name
```

Since the transformation passed to `everywhere` is applied to multiple different types, we use `mkT` as an adapter to generalize `f` from a transformation on `HsName` to a transformation on any type.[2] When applied to a value that is an `HsName`, `mkT f` is just the function `f`, but on other types it is the identity function.

TYB draws inspiration from SYB and exposes a similar interface. SYB performs the type dispatch needed by generic programming at runtime. TYB achieves the convience of SYB while having the runtime performance of handwritten code by moving this type dispatch to compile time. It does this using Template Haskell splices that generate traversals customized to the particular types that they traverse. At an intuitive level, one could consider the code generated by TYB as the result of partially evaluating or inlining SYB.

Most programs written with SYB require only minor modification to use TYB. In TYB, the equivalent of `prefixNames`$_{syb}$ is written:

```
prefixNames x =
  $(everywhere (mkT 'f) [|t HsModule |]) x where
    f :: HsName → HsName
    f name = prefixName "_" name
```

Here, `everywhere` is inside a splice and thus executes at compile time. It generates a Haskell code fragment for traversing an `HsModule` and the compiler compiles that fragment in place of the splice. The types for `everywhere` and TYB's other core functions are shown in Figures 2 and 3

Aside from the additional syntax needed by Template Haskell, the `everywhere` function in TYB takes an extra argument that is not in the SYB version. It specifies the type that `everywhere` traverses. In this case, the argument is $[\![_t$ `HsModule` $]\!]$ so the traversal is over an `HsModule` and any values inside an `HsModule` regardless of their types. Systems like SYB can use the type expected by the surrounding code to determine the type over which to traverse. But due to limitations imposed by Template Haskell, code inside a splice does not know the type expected outside the splice. Thus, in TYB we have to explicitly specify the type over which to traverse.

For this small increase in code complexity, the performance improvements are dramatic. As shown in Section 5, the SYB version takes ten times longer than a handwritten traversal, but the TYB version matches the speed of a handwritten traversal.

TYB also defines both a monadic traversal (`everywhereM`) and a query traversal (`everything`). These parallel their counterparts

[2] Technically the transformation is only over instances of the `Typeable` class, but for the purposes of this paper, this is a minor point.

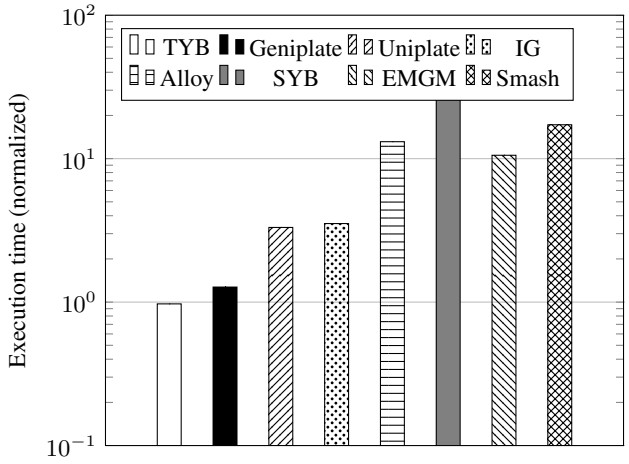


Figure 1: Runtime relative to handwritten code (geometric mean).

in SYB. For example, consider the task of freshening all of the identifiers in an `HsModule`. Suppose we have a function `freshenName` that has type `HsName → Fresh HsName` for some monad `Fresh`. With SYB, to apply `freshenName` to every identifier, we write:

```
freshenNames_syb x = everywhereM (mkM freshenName) x
```

With TYB, this is equally easy. The main differences are the use of Template Haskell syntax and the need to pass the argument to `everywhereM` that specifies the type over which `freshenNames` traverses. It is thus written:

```
freshenNames x =
  $(everywhereM (mkM 'freshenName) ⟦t HsModule ⟧) x
```

As a final example, consider a query traversal that lists all `HsNames` contained in an `HsModule`. With SYB this is written:

```
listNames_syb x = everything (++) (mkQ [] f) x where
  f :: HsName → [HsName]
  f x = [x]
```

Again, using TYB for this query requires only minor modifications relative to the SYB version and is written:

```
listNames x = $(everything ⟦(++)⟧  (mkQ ⟦[]⟧ 'f)
                           ⟦t HsModule ⟧) x where
  f :: HsName → [HsName]
  f x = [x]
```

In both `freshenNames` and `listNames`, `mkM` and `mkQ` serve the same role as `mkT` does in `prefixNames`, but for monadic transformations and queries respectively.

With query traversals, however, we have a few extra arguments. The `(++)` passed to `everything` specifies how to combine query results from multiple sub-terms. The `[]` passed to `mkQ` specifies the result for a sub-term that is not an `HsName`.

The TYB versions of these traversals perform the same calculations as the SYB versions, but the TYB versions complete their calculations over ten times faster.

Most of the operators provided by TYB follow this same pattern. A traversal written with SYB can easily be converted to TYB by the addition of splices around generic operators, and minor modifications to their arguments.

TYB offers a simple interface to generic programming that eliminates the need for handwritten boilerplate code. But unlike most other generic-programming systems, it is just as fast as handwritten code. In Section 5, we benchmark TYB against both handwritten code and several other generic-programming systems. Figure 1 summarizes the results and shows the average performance of each system relative to handwritten code. TYB often outperforms other systems by an order of magnitude or more. The only other system that comes close is Geniplate [Augustsson 2011], which uses many of the same techniques as TYB but has a more restrictive interface, as we discuss later in Section 6.2.

Primitives

```
constructorsOf :: Type → Q (Maybe [(Name, [Type])])
typeOfName :: Name → Q Type
expandType :: Type → Q Type
```

Value deconstruction

```
thfoldl :: (                Q Exp {- c (d → b) -}
          → Type  {- d -} → Q Exp {- d -}
          →                 Q Exp {- c b -} )
         → (Q Exp  {- g -} → Q Exp {- c g -} )
         →  Q Type {- a -} → Q Exp {- a → c a -}

thcase :: (Q Exp                  {- a → b → ... → t -}
          → [(Type, Q Exp)]       {- a, b, ... -}
          → Q Exp                 {- c t -} )
         → Q Type {- t -} → Q Exp {- t → c t -}
```

One-layer traversal

```
thmapT :: (Type {- b -} → Q Exp {- b → b -} )
       → Q Type {- a -} → Q Exp {- a → a -}
thmapM :: (Type {- b -} → Q Exp {- b → m b -} )
       → Q Type {- a -} → Q Exp {- a → m a -}
thmapQ :: (Type {- b -} → Q Exp {- b →  r  -} )
       → Q Type {- a -} → Q Exp {- a → [r] -}
```

Figure 2: Core functions provided by TYB (part 1).

4. Implementation

The technique that makes TYB execute efficiently is that it elaborates generic operations at compile time and, in the process, generates code specialized to particular types. This specialized code contains no generic parts and thus executes efficiently at runtime.

We use the splice form ($\$(e)$) from Template Haskell for this compile-time code generation. In the `prefixNames` example from Section 3, the call to `everywhere` is executed at compile time since it is inside a splice. Thus, `everywhere` does not itself traverse a value of type `HsModule`. Rather, it generates a Template Haskell `Exp` for code that, when executed at runtime, will traverse a value of type `HsModule`.

The core functions provided by TYB are listed in Figures 2 and 3. Several of them closely parallel the corresponding functions from SYB, which are shown in Figure 4 for reference. The compile time versus runtime split does change the types of these functions, however. With SYB, the functions manipulate values, but with TYB the functions manipulate AST fragments. In Figures 2 and 3, the signatures are annotated with comments denoting the type for each AST fragment. For `Exp`, this is the type of the expression. For `Type`, this is the value of the type. For `Name`, this is the type of the value bound to that identifier. For example, the first argument to `everywhere` has type `Type → Q Exp`, and given the `Type` that is the AST representation of a particular type `b`, the returned AST fragment should represent a function that expects a value of type `b` and returns a transformed value of type `b`.

Recursive traversal

```
everywhere :: (Type {-b-} → Q Exp {-b → b-} )
           → Q Type {-a-} → Q Exp {-a → a-}
everywhereFor ::                Name  {-b → b-}
              → Q Type {-a-} → Q Exp {-a → a-}
everywhereBut :: ( Type {-a-} → Q Bool)
              → ( Type {-a-} → Q Exp {-a → a-} )
              → Q Type {-a-} → Q Exp {-a → a-}
everywhereM :: (Type {-b-} → Q Exp {-b → m b-} )
            → Q Type {-a-} → Q Exp {-a → m a-}
everything :: Q Exp {-r → r → r-}
           → ( Type {-b-} → Q Exp {-b → r-} )
           → Q Type {-a-} → Q Exp {-a → r-}
```

```
inType :: Type → Type → Q Bool
memoizeExp ::
  ((Type → Q Exp) → Type → Q Exp) → Type → Q Exp
```

Dispatch

```
mkT ::               Name {-b → b-}
    → Type {-a-} → Q Exp {-a → a-}
mkM ::               Name {-b → m b-}
    → Type {-a-} → Q Exp {-a → m a-}
mkQ :: Q Exp {-r-} → Name {-b → r-}
    → Type {-a-} → Q Exp {-a → r-}
```

```
extN :: (Type {-c-} → Q Exp {-c → h c-} )
     →               Name  {-b → h b-}
     →  Type {-a-} → Q Exp {-a → h a-}

extE :: (  Type {-c-} → Q Exp {-c → h c-} )
     → (Q Type {-b-} , Q Exp {-b → h b-} )
     →    Type {-a-} → Q Exp {-a → h a-}
```

Figure 3: Core functions provided by TYB (part 2).

Primitives

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

One-layer traversal

```
gmapT :: Data a => (∀ b. Data b => b → b) → a → a
gmapQ :: Data a => (∀ d. Data d => d → u) → a → [u]
gmapM :: (Data a, Monad m) =>
         (∀ d. Data d => d → m d) → a → m a
```

Recursive traversal

```
everywhere :: (∀ b. Data b => b → b)
           → (∀ a. Data a => a → a)
everywhereM :: (Monad m) =>
               (∀ b. Data b => b → m b)
            → (∀ a. Data a => a → m a)
everything :: (r → r → r)
           → (∀ b. Data b => b → r)
           → (∀ a. Data a => a → r)
```

Dispatch

```
mkT :: (∀ b. Typeable b => b → b)
    → (∀ a. Typeable a => a → a)
mkQ :: r → (∀ b. Typeable b => b → r)
         → (∀ a. Typeable a => a → r)
mkM :: Monad m => (∀ b. Typeable b => b → m b)
                → (∀ a. Typeable a => a → m a)

extT :: (Typeable b, Typeable a) =>
        (a → a) → (b → b) → a → a
extQ :: (Typeable b, Typeable a) =>
        (a → q) → (b → q) → a → q
extM :: (Typeable b, Typeable a, Monad m) =>
        (a → m a) → (b → m b) → a → m a
```

Figure 4: Core functions provided by SYB.

4.1 From `gfoldl` to `thfoldl`

While `thfoldl` is not the most primitive operator in TYB, it provides a convenient place from which to start and has an interface that should be familiar to users of SYB.

In SYB, the `gfoldl` method of the `Data` class follows a predictable structure that can be seen by examining the definition of `gfoldl` for the following `List` type.

```
data List = Nil | Cons Int List

instance Data List where
  gfoldl k z = λe → case e of
    Nil       → z Nil
    Cons x xs → z Cons `k` x `k` xs
```

The pattern here is that for any object, such as `Cons x xs`, the `z` function is applied to the constructor, `Cons`, and the `k` function is used to apply the result to the constructor's arguments, `x` and `xs`.

In TYB, the equivalent operation, `thfoldl`, runs at compile time and generates an AST fragment based on compile-time information about the constructors of a given type. It thus avoids the runtime overheads involved in using `gfoldl`. While we do not give an implementation for `thfoldl` until the end of Section 4.3, the pattern that it follows can be seen by examining what it generates for the `List` type. The following reduction is not quite right as it glosses over the type of `k`, but it gives the right intuition, and we will correct it in a moment.

```
thfoldl k z ⟦t List ⟧ ↦
  ⟦λe → case e of
    Nil       → $(z ⟦ Nil  ⟧)
    Cons x xs → $(z ⟦ Cons ⟧ `k` ⟦ x ⟧ `k` ⟦ xs ⟧)⟧
```

The preceding reduction glosses over the type of `k`. In SYB, the second argument of the `k` parameter is qualified by the `Typeable` class indirectly through the `Data` class. This allows `k` to inspect the type of its second argument and compute different results based on that type. In TYB, the `thfoldl` function also allows `k` to inspect the type of the constructor's argument, but rather than using the `Typeable` class, the `Type` of the argument is directly passed with the argument. Thus, the preceding reduction is actually as follows.

```
thfoldl k z ⟦t List ⟧ ↦
  ⟦λe → case e of
    Nil       → $(z ⟦ Nil ⟧)
    Cons x xs → $(k (k (z ⟦ Cons ⟧)
                       ⟦t Int ⟧ ⟦ x ⟧)
                    ⟦t List ⟧ ⟦ xs ⟧)⟧
```

Owing to the additional argument to `k`, we do not use infix notation here, but it is otherwise the same as the previous reduction.

Implementation:

Reduction on $[\![_t$ `List` $]\!]$:

`thfoldl`:

```
thmapQ f t₀ = thfoldl k z t₀ where
  z c     = [[]]
  k c t x = [$(c) ++ [$(f t) x]]
```

```
thmapQ f [[t List]] ↦
  [λe → case e of
    Nil       → []
    Cons x xs → ([] ++
                 [$(f [[ Int ]]) x]) ++
                 [$(f [[ List ]]) xs]]
```

`thcase`:

```
thmapQ f t₀ = thcase g t₀ where
  g c []           = [[]]
  g c ((t, x) : xs) = [$(f t) $(x) :
                       $(g c xs)]
```

```
thmapQ f [[t List]] ↦
  [λe → case e of
    Nil       → []
    Cons x xs → $(f [[t Int]]) x :
                $(f [[t List]]) xs :
                []]
```

Figure 5: Implementation options for `thmapQ` and their resulting reductions.

4.2 One-layer traversal

With `thfoldl`, we can build one-layer traversals that apply a particular operation to the immediate descendants of an object. In SYB, an example is `gmapT`, which applies a transformation to the immediate descendants of an object. For the `List` type, `gmapT` behaves as follows.

```
gmapT f ↦ λe → case e of
              Nil       → Nil
              Cons x xs → Cons (f x) (f xs)
```

The TYB version of this is `thmapT` and is similar except that it generates code that does the transformation rather than doing the transformation itself. In addition, `f` takes an argument indicating the type of the constructor's argument to which it is applied. On the `List` type it reduces as follows.

```
thmapT f [[t List]] ↦
  [λe → case e of
    Nil       → Nil
    Cons x xs → Cons ($(f [[t Int]]) x)
                     ($(f [[t List]]) xs)]
```

Note that the type of the `Q Exp` returned by `f` must vary according to the `Type` given to `f`. Given the `Type` for type `t`, `f` should return a `Q Exp` for an expression of type `t → t`.

We implement `thmapT` in terms of `thfoldl` as follows.

```
thmapT f t₀ = thfoldl k z t₀ where
  z c     = c
  k c t x = [$(c) ($(f t) $(x))]
```

Other one-layer traversals such as `thmapQ` and `thmapM` can also be implemented in terms of `thfoldl`, but the generated code is not as efficient as we desire. The remainder of this subsection explores how these inefficiencies occur. The next subsection shows how to eliminate them by generalizing from `thfoldl` to `thcase`.

To see how these inefficiencies occur, consider the `thfoldl`-based implementation of `thmapQ` in Figure 5. The left column of that figure contains the implementation for an arbitrary type t_0, and for illustration purposes, the right column contains reductions for $[\![_t$ `List` $]\!]$.

The `thmapQ` function returns a list of the results of applying `f` to each immediate child while preserving the order of the children. So, for example, the result from the leftmost child should be leftmost in the returned list. Since `thfoldl` is a left fold instead of a right fold, however, the implementation must use list append, `(++)`, instead of the more efficient list cons, `(:)`.

This inefficiency may be small as the lists involved are bounded in length by the number of arguments in a constructor and most constructors have only a few arguments. Nevertheless, barring optimizations by the compiler, it is less efficient than handwritten code.

The `thfoldl`-based implementation of `thmapM` shown in Figure 6 has a similar problem. It applies a monadic transform to each immediate child. The `ap` function from `Control.Monad` is used to apply a monad containing the constructor to the monadic computation for each of its arguments and has type:

```
ap :: (Monad m) => m (a → b) → m a → m b
```

The inefficiency of this implementation may not be immediately obvious so consider the reduction of `thmapM f` $[\![_t$ `List` $]\!]$ after inlining `ap` and translating monadic binds to *`do`* notation:

```
thmapM f [[t List]] ↦
  [λe → case e of
    Nil       → return Nil
    Cons x xs → do c₁ ← do c₀ ← return Cons
                           x₀ ← $(f [[t Int]]) x
                           return (c₀ x₀)
                   x₁ ← $(f [[t List]]) xs
                   return (c₁ x₁)]
```

This code is semantically correct, but the `Cons` clause involves three returns and four monadic binds. In the next subsection we move from a `thfoldl`-based implementation to a `thcase`-based implementation and show how this same code can be implemented with only one return and two monadic binds.

For both `thmapQ` and `thmapM`, the inefficiency is due to `thfoldl` being a left fold when the code being generated is more naturally expressed as a right fold. This forces us to use `(++)` in the case of `thmapQ` and `ap` in the case of `thmapM`.

Left folds are often equivalent to right folds so it is tempting to think that given an appropriate trick, `thfoldl` might still be sufficient. However, that equivalence holds only when the type of the fold is sufficiently general. With `thfoldl`, the `z` and `k` arguments cannot have such a general type and must return `Q Exp`. This is because `thfoldl` is not just a fold. Rather, it generates code that performs a fold at runtime. As part of that, it performs, at compile time, folds over the arguments of the constructors, but the results of those folds are not simply returned. Instead, they are placed inside the `Exp` for a *`case`* statement that then discriminates between constructors at runtime.

In SYB, the `gfoldl` function does not have this problem as it is simply a fold over a particular value. Thus the type of `gfoldl` is general enough to thread extra information through the fold.

Implementation:

```
thfoldl:  thmapM f t₀ = thfoldl k z t₀ where
            z c     = ⟦ return $(c) ⟧
            k c t x = ⟦ $(c) `ap` $(f t) $(x) ⟧

thcase:   thmapM f t₀ = thcase g t₀ where
            g c []            = ⟦ return $(c) ⟧
            g c ((t, x) : xs) = ⟦ $(f t) $(x) >>= λx' →
                                  $(g ⟦ $(c) x' ⟧ xs) ⟧
```

Reduction on $\llbracket_t$ `List` $\rrbracket$:

```
thmapM f ⟦t List ⟧ ↦
  ⟦λe → case e of
    Nil       → return Nil
    Cons x xs → return Cons `ap`
                  $(f ⟦t Int  ⟧) x `ap`
                  $(f ⟦t List ⟧) xs ⟧

thmapM f ⟦t List ⟧ ↦
  ⟦λe → case e of
    Nil       → return Nil
    Cons x xs → do x'  ← $(f ⟦t Int  ⟧) x
                   xs' ← $(f ⟦t List ⟧) xs
                   return (Cons x' xs')⟧
```

Figure 6: Implementation options for `thmapM` and their resulting reductions.

Since efficiency is a prime consideration in the design of TYB, we want to avoid the inefficiencies forced by the interface of `thfoldl`. Compiler optimizations might eliminate some inefficiencies, but we should not rely too heavily on that. For example, the monadic binds in `thmapM` might not be inlineable or the compiler might not be able to determine whether their implementation follows the monad laws needed to eliminate these inefficiencies. It is better to generate efficient code in the first place. The next subsection shows how to do this.

4.3 From `thfoldl` to `thcase`

As shown in the previous subsection, while an efficient version of `thmapT` can be implemented in terms of `thfoldl`, the same is not true of `thmapQ` and `thmapM`. The structure of `thfoldl` imposes a structure on the generated code that leads to runtime inefficiency.

To resolve this, we generalize the interface of `thfoldl`. The essential task of `thfoldl` is the construction of an `Exp` containing a *case* statement. With `thfoldl`, the user specifies each clause of the *case* statement in terms of a fold over the arguments of a given constructor. Instead of having the user specify clauses in terms of folds, there is no reason not to simply pass the constructors and the lists of their arguments to a user supplied function.

The `thcase` function provides such an interface. It takes a function `g` and a type $\texttt{t}_0$. It constructs a *case* statement appropriate for $\texttt{t}_0$ and, for each constructor of $\texttt{t}_0$, calls `g` with the constructor and a list of its arguments and argument types. The arguments are provided as a simple list, so the folding strategy is left up to the user. For example, with the `List` type it behaves as follows.

```
thcase g ⟦t List ⟧ ↦
  ⟦λe → case e of
    Nil       → $(g ⟦ Nil  ⟧ [])
    Cons x xs → $(g ⟦ Cons ⟧ [
                    (⟦t Int  ⟧, ⟦ x  ⟧),
                    (⟦t List ⟧, ⟦ xs ⟧)])⟧
```

For primitive types[3] such as `Int`, `thcase` passes the value itself as the first argument to `g`. Effectively, the value is the constructor. For example with `Int`, it behaves as follows.

```
thcase g ⟦t Int ⟧ ↦ ⟦λe → $(g ⟦ e ⟧ [])⟧
```

[3] We consider `Int` a primitive type despite being it decomposable into an `I#` constructor on an `Int#`. In a recursive traversal, this prevents the kinding error of instantiating a polymorphic function on a type of kind `#` (e.g., instantiating `return` at `Int#`). It is an open question how best to give the user control over what constitutes a primitive type.

The implementation of `thcase` is shown in Figure 7. At its core, it uses the `constructorsOf` primitive provided by TYB to inspect the constructors for the given type and build an appropriate *case* statement. For non-primitive types, `constructorsOf` returns `Just` of a pair of the constructor name and its argument types. For primitive types, `constructorsOf` returns `Nothing`. For example, we have the following reductions for `Int` and `List`, which return the constructors of `Int` and `List`, respectively.

```
constructorsOf ⟦t Int  ⟧ ↦ return Nothing
constructorsOf ⟦t List ⟧ ↦
  do int  ← ⟦t Int  ⟧
     list ← ⟦t List ⟧
     return (Just [('Nil , []),
                   ('Cons, [int, list])])
```

The `newName` and `varE` functions in Figure 7 are provided directly by Template Haskell. The `newName` function creates a fresh `Name`, and `varE` takes a `Name` and returns a `Q Exp` that is a variable reference to that `Name`.

We are slightly abusing Template Haskell syntax in Figure 7. Template Haskell neither supports the splicing of clauses into the body of a *case* nor provides a quotation syntax for clauses. Thus, instead of the quotations used in Figure 7, the actual implementation of `thcase` uses the AST constructors provided by Template Haskell. In Figure 7, we use quotations simply because they are easier to read and understand.

The `constructorsOf` function is implemented in terms of the `reify` function provided by Template Haskell. It is used to interrogate the compiler about the constructors of a particular type. Though the intuition behind `constructorsOf` is simple, the implementation needs to handle a variety of complications including type synonyms, type substitutions that arise due to type constructor application, and the various methods of defining types (e.g., *data*, *newtype*, records, etc.). None of these are theoretically deep complications, but the necessary code is verbose and not particularly interesting. Thus we omit the implementation of `constructorsOf` from this paper.

With the `thcase` function, efficient versions of `thmapQ` and `thmapM` are trivial to define as shown in Figures 5 and 6. As shown in the right hand columns, the generated code for types such as `List` is exactly what one would expect in a handwritten traversal.

Finally, note that `thfoldl` is easily defined using `thcase`:

```
thfoldl k z t = thcase g t where
  g ctor args = foldl (uncurry . k) (z ctor) args
```

```
thcase g t₀ = do
  cs ← constructorsOf =<< t₀
  case cs of
    Nothing  → ⟦λe → $(g ⟦ e ⟧ [])⟧
    Just cs' → ⟦λe → case e of $(mapM clause cs')⟧
  where clause (name, types) = do
          args ← mapM (λt → newName "arg") types
          let argsE = zip types (map varE args)
          in ⟦ $(name) $(args) → $(g name argsE) ⟧
```

Figure 7: The implementation of `thcase`.

4.4 Recursive traversal

Recursive traversals are easily defined in terms of single-layer traversals. For example, the `everywhere` function applies a transformation at every node in a tree. It is implemented by applying the transformation to the current node and using `thmapT` to apply `everywhere` to each child. This results in recursively applying the transformation to every descendant. Depending on whether the transformation or the `thmapT` is applied first, this results in either a bottom-up or a top-down traversal. The code for a naive implementation of such a function is:

```
everywhere f t = ⟦λx →
  $(f t) ($(thmapT (everywhere f . return) t) x)⟧
```

This naive implementation has a flaw, however. At compile time, it will loop forever if it is applied to a recursive type such as `List`. This is because it recurses over the structures of types instead of the structures of values. For example, on the `List` type, the infinite loop is caused by the `List` in the `Cons` constructor. When `everywhere` is called on `List`, it thus recursively calls `everywhere` on `List`. That recursive call also recursively calls `everywhere` on `List`, and so on. Where systems like SYB recurse infinitely on recursive or cyclic *values*, this naive implementation of `everything` recurses infinitely on recursive *types*.

Since recursive types are quite common in Haskell, this is a significant problem. Fortunately, it is easily solved by memoizing the compile-time calls to `everywhere`. Each call to `everywhere` on a particular type generates the same `Exp` so rather than generating duplicate expressions, we bind the expression to a variable that we then reference as needed. For example, if we memoize all the types in `List` (i.e., `List` and `Int`), the resulting reduction is:

```
everywhere f ⟦t List ⟧ ↦
  ⟦ let mem_Int  = λe → $(f ⟦t Int  ⟧) e
        mem_List = λe → $(f ⟦t List ⟧) (case e of
                     Nil       → Nil
                     Cons x xs → Cons (mem_Int  x)
                                      (mem_List xs))
    in mem_List ⟧
```

Here $\texttt{mem}_{Int}$ is the memoization at `Int` and $\texttt{mem}_{List}$ is the memoization at `List`. The recursive structure of the `List` type is manifest in the recursive structure of $\texttt{mem}_{List}$.

On recursive or cyclic *values*, this will still recurse infinitely at runtime just as handwritten code would, but memoizing eliminates the problems with recursive *types*.

The `memoizeExp` function provided by TYB implements this memoization for general types. The `memoizeExp` function takes as its first argument the function to be memoized. It then passes a memoized version of that function as the first argument to the function itself. Essentially, it is a fixed-point operation but in the process of "tying the knot" it adds memoization.

```
everywhere f t₀ = t₀ >>= memoizeExp rec where
  rec r t = ⟦λe → $(f t)
                  ($(thmapT r (return t)) e)⟧

everywhereM f t₀ = t₀ >>= memoizeExp rec where
  rec r t = ⟦λe → $(f t) =<<
                  ($(thmapM r (return t)) e)⟧

everything o f t₀ = t₀ >>= memoizeExp rec where
  rec r t = ⟦λe → foldl $(o) ⟦ $(f t) e ⟧
                  ($(thmapQ r (return t)) e)⟧
```

Figure 8: Implementations for the recursive traversal functions.

The `everywhere`, `everywhereM`, and `everything` functions provided by TYB are implemented using `memoizeExp` as shown in Figure 8. These implementations handle recursive types without recurring infinitely at compile time.

The implementation of `memoizeExp` is a standard memoization keyed by the `Type` that is being traversed and thus its implementation is omitted from this paper. However, there is a caveat to the memoization process: polymorphically recursive, non-regular types can still lead to infinite recursion at compile time. For example, consider the following type:

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

The `T Int` type recursively contains `T (Int, Int)` which recursively contains `T ((Int, Int), (Int, Int))` and so on. Since these are all different types, memoization cannot reuse the expressions generated for each of these types and thus an infinite recursion can occur at compile time. The system presented in this paper does not provide a solution for this situation, unlike SYB, which handles it just fine.

Finally, note that these traversals cannot operate on types containing variables. For example, `$(everything f ⟦t ∀ a. [a] ⟧)` is a compile time error as `thcase` has no way to determine the constructors of the type `a`.

4.5 Selective traversal

As discussed later in Section 5.4, many traversals benefit significantly from skipping the parts of a traversal that reach a value having a type that cannot contain any types of interest. For example, given that an `HsName` cannot occur in a `String`, there is no point in traversing the contents of a `String` when looking for an `HsName`. This is easily implemented by passing a predicate to the traversal that tells it when to stop. Figure 9 shows the implementation of `everywhereBut` as an example of this. In general, the predicate can be anything, but a particularly useful one is `inType`, which is also shown[4] in Figure 9. It uses the `expandType` primitive provided by TYB to expand type synonyms and remove kind annotations before comparing the current type, `t`, against the target type, `s`. This expansion ensures that types like `String` and `[Char]` are considered equal. If `t` has already been seen, then we are at a cycle in the type structure and `inType` returns false. If `s` and `t` match, then `inType` returns true. In all other cases, `inType` uses `constructorsOf` to get the potential types of immediate subterms and recurses over them. Thus it returns true if a term of type `t` can contain a term of type `s` and returns false if not.

[4] The practical implementation of `inType` is a bit more careful about keeping track of seen types and avoids the exponential explosion latent in this version by storing the seen types in an `IORef` embedded in the `Q` monad.

```
everywhereBut q f t₀ = t₀ >>= memoizeExp rec where
  rec r t = do
    b ← q t
    if b then ⟦λe → e ⟧
         else ⟦λe → $(f t)
                    ($(thmapT r (return t)) e)⟧

inType s t = do s' ← expandType; rec [] s' t where
  rec seen s t = do
    t' ← expandType t
    if t' `elem` seen then return False
    else if t' ≡ s then return True
    else do cs ← constructorsOf t'
            case cs of
              Nothing  → return False
              Just cs' → check (t' : seen) s
                                 (concatMap snd cs')
  check seen s []       = return False
  check seen s (t : ts) =
    do t' ← rec seen s t
       if t' then return True else check seen s ts

everywhereFor name =
  do t ← typeOfName name
     everythingBut (liftM not . inType (arg t))
  where arg :: Type → Type
        arg (AppT (AppT ArrowT t) _) = t
        arg (ForallT _ _ t)          = arg t
```

Figure 9: Selective traversal implementations.

To further simplify the task for the user, we can combine the `typeOfName` primitive provided by TYB with `everywhereBut` and `inType`. The result is `everywhereFor` in Figure 9. The `typeOfName` primitive returns the type of the binding for a particular `Name`. The `everywhereFor` function inspects the result of `typeOfName` using `arg` and based on that makes an appropriate call to `everywhereBut` and `inType`.

These are just a few of the traversals easily expressed by TYB. The full API includes functions for left and right biased accumulation, strict traversals and many others. As shown in these examples, however, it is straightforward for users to write additional traversals if the existing ones do not meet their needs.

4.6 Dispatch

The `f` function that the user passes to recursive traversals takes the type of a particular value in the traversal and returns an AST fragment appropriate for that type. The dispatch operators provide convenient shortcuts for the common cases.

The `mkT`, `mkM`, and `mkQ` functions expect the `Name` of a function and, based on the type of that function, return an appropriate value for the `f` argument of traversals. These functions expect the name of a transform function, a monadic transform function, or a query function respectively. They return a function that compares the type of the named function to see whether it is appropriate for the type to which `f` is applied. If it is, then the returned `Exp` is a reference to the named function. If not, then the returned `Exp` is a reference to a neutral element. For `mkT`, this neutral element is `id`. For `mkM`, it is `return`. For `mkQ`, it is provided by the user.

The `extN` function works similarly except that instead of returning a neutral element when not returning a reference to the named function, it delegates to another function provided by the user. This makes it possible to chain together multiple functions that each handle their own type. Unlike in SYB, where different operators are used depending on whether the function is monadic, a query, or neither, `extN` serves all three purposes in TYB.

The most general dispatch operator is `extE` which takes a `Type` directly rather than extracting it from the type of the function specified by a `Name`.

The implementation of all these functions is straightforward given the `typeOfName` and `expandType` primitives and are omitted from this paper.

4.7 Summary

The TYB system has three fundamental primitives that deal with the complexities of Template Haskell. They are `constructorsOf`, `typeOfName` and `expandType`. The higher-layers in the system are straightforwardly implemented in terms of these primitives, and though it does require thinking in terms of meta-programming, it is relatively easy to extend the system with new traversals and operations.

5. Performance

To judge the relative performance of TYB, we implemented several generic-programming tasks using both TYB and other generic-programming systems. We then compared these implementations to handwritten implementations. We selected benchmarks implementable by all of the systems as we aim to measure performance, and not features.

Among the benchmarked systems, TYB and Geniplate [Augustsson 2011] use Template Haskell to generate traversals at compile time, while the other systems use more traditional generic-programming techniques. Geniplate shares the performance benefits of Template Haskell that TYB has, but presents the user with a much more limited interface as we discuss later in Section 6.2. The benchmark results are shown in Figure 10 and are normalized relative to handwritten code. In general, TYB and Geniplate performed several times faster than other systems. The results are discussed in more detail at the end of this section.

5.1 Benchmarks

List manipulation The *map* and *sum* benchmarks implement `map` and `sum` from Haskell's `Prelude` but in a generic-programming style and only for `Int` lists. For the handwritten version of these benchmarks, we use `map` and `sum` from the GHC `Prelude`. The list type is small enough that there is little need for a generic implementation of these functions, but these benchmarks have the advantage of having preexisting, efficient, standard, handwritten implementations.

GPBench Rodriguez et al. [2008] present a benchmarking suite for generic-programming systems called GPBench. While its primary focus is evaluating the features of generic-programming systems, three of the benchmarks evaluate performance. They are *selectInt*, *rmWeights*, and *geq*. In the versions that we use, they all operate over the following type of weighted trees.

```
data WTree a w = Leaf a
               | Fork (WTree a w) (WTree a w)
               | WithWeight (WTree a w) w
```

The *selectInt* benchmark collects a list of all values of type `Int` occurring in a `WTree Int Int`. It is a query traversal. The *rmWeights* benchmark traverses a `WTree Int Int` replacing every `WithWeight t w` with `t`. It is a transform traversal. The *geq* benchmark traverses two `WTree` objects checking that they are equal and is a twin or parallel traversal. This sort of traversal is not supported by several of the benchmarked systems and is omitted from our

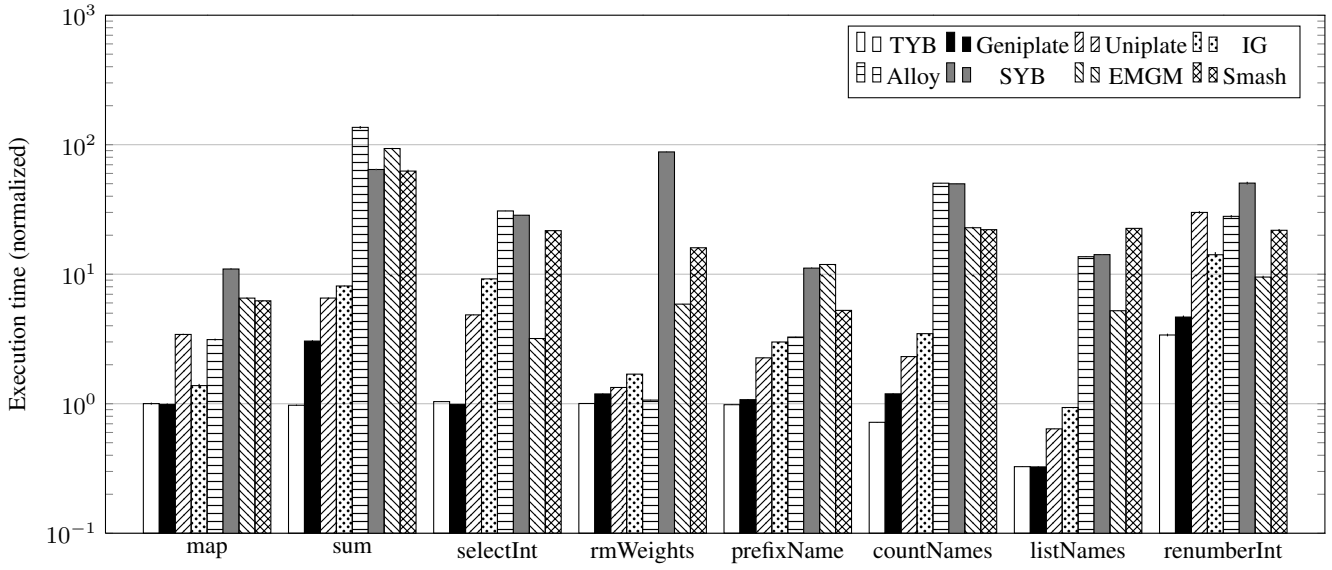


Figure 10: Benchmark running time relative to handwritten code.

benchmarking tests. For these benchmarks the handwritten implementations are taken directly from GPBench.

Notably absent from GPBench is a benchmark for measuring the performance of monadic traversal so we add one benchmark not in the original GPBench that we call *renumberInt*. The *renumberInt* benchmark traverses a `WTree Int Int` and uses a state monad to replace each `Int` with a unique value.

AST manipulation Both the list manipulation and GPBench benchmarks operate on fairly small types. The list type has only two constructors. The `WTree` type has only three constructors. Both are recursive in only one type.

To evaluate the performance on larger types, we include benchmarks on a real-world AST. Specifically, we use the AST types from `Language.Haskell.Syntax`, which includes types for expressions, declarations, statements, patterns and many other Haskell forms. In total, over 30 different types with over 100 constructors are involved, in addition to types from the `Prelude` such as booleans, strings, tuples, lists, and so forth.

The AST object we use is the parsed, preprocessed source for `Data.Map` from the source for GHC version 7.0.3. The preprocessed source file is 2,164 lines long, and the resulting AST has 74,921 nodes.

The *prefixName* benchmark finds every `HsName` object and prefixes it with an underscore (`_`). The *countName* benchmark counts the number of `HsName` objects in the AST. The *listName* benchmark returns a list of all `HsName` occurring in the AST.

For these benchmarks, the handwritten version is written in a straightforward, mechanical style. No attempt was made to tune the boilerplate portion of the code. Base cases, such as `String`, are not traversed by the handwritten implementation, but we did no further tuning of traversals as that would be prohibitively time consuming and error prone.

5.2 Technical specifications

Aside from the handwritten and TYB versions, each benchmark was also implemented using

- Geniplate version 0.6.0.0 [Augustsson 2011],
- Uniplate[5] [Mitchell and Runciman 2007],
- Instant Generics (IG) version 0.3.4 [Chakravarty et al. 2009; van Noort et al. 2008],
- Alloy version 1.0.0 [Brown and Sampson 2009],
- Scrap Your Boilerplate (SYB) version 0.3.6 [Lämmel and Peyton Jones 2003],
- Extensible and Modular Generics for the Masses (EMGM) version 0.4 [Oliveira et al. 2006], and
- Smash Your Boilerplate (Smash) [Kiselyov 2006].

Scrap Your Boilerplate was included because it is the most well known and widely used generic-programming system for Haskell. EMGM, Smash, and Uniplate were included because Rodriguez et al. [2008] identified them as the best performing of the generic-programming systems that they surveyed. Alloy and IG are included as they were published after the work by Rodriguez et al. and thus were not included in their survey but report good performance. Geniplate was included because it also uses Template Haskell to generate traversals at compile time.

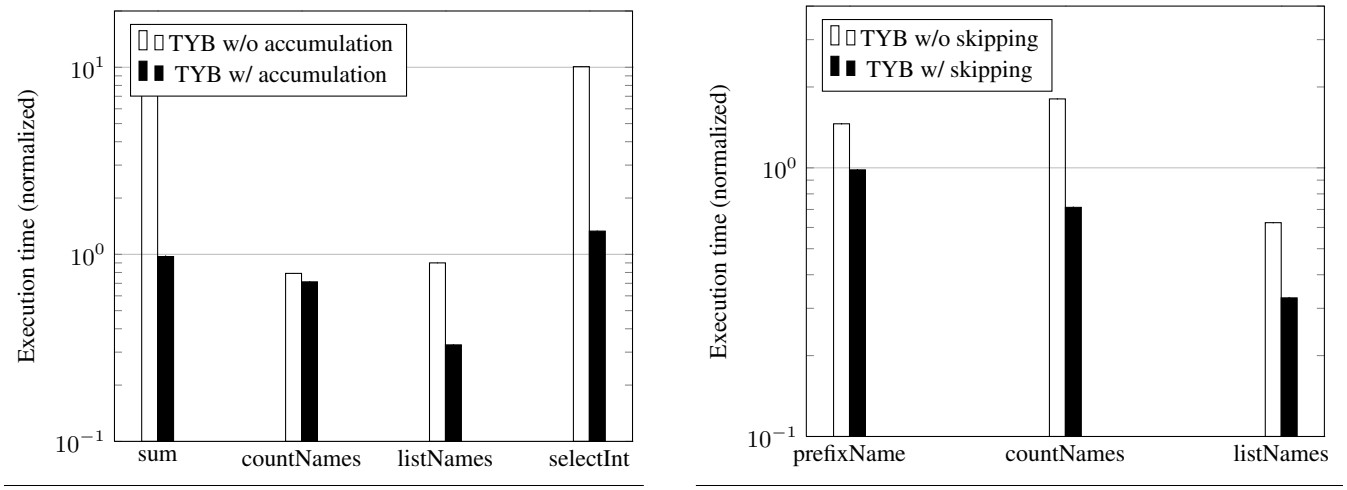
We tested both the "with overlapping" and "without overlapping" variants of Alloy, but they produced essentially identical results. We used the "direct" variant of Uniplate as that is the fastest variant. We used the version of Smash available from the GPBench repository (`http://code.haskell.org/generics/`).

We benchmarked with GHC 7.0.3 using `-O2` on a 3.2 GHz, 64-bit Xeon with 4 GB of RAM running Ubuntu Linux 11.10.

5.3 Results

The results of these benchmarks are shown in Figure 10. All times are normalized relative to the performance of the handwritten implementations. Times are calculated using *Criterion* [O'Sullivan 2012] and are the mean of several thousand executions. Error bars

[5] Uniplate's authors graciously provided us with a preview version that fixes some performance bugs identified by these benchmarks. The results reported here are from that version. A public release with these fixes should be available soon.



(a) Accumulating traversal.

(b) Selective traversal.

Figure 11: Benchmark results for alternative traversal strategies.

are one standard deviation, but in most cases they are too small to see on the chart.

Care must be taken when interpreting these results. While the benchmarks were chosen from standard generic-programming tasks, they do not necessarily represent a wide-enough cross section from which to draw any conclusions beyond broad trends. This is a particularly important caveat as both Rodriguez et al. [2008] and Brown and Sampson [2009] report that the relative performance of many systems varies widely depending on the details of the benchmark.

Nevertheless, a few trends are clear. The reputation of SYB for being slow is well deserved. In every benchmark, it is at least ten times slower than handwritten code. In some cases, it is almost one hundred times slower. Smash exhibits a performance pattern similar to that of SYB. Uniplate and IG perform considerably better but are still often five to ten times slower than handwritten code. Alloy comes close to matching the performance of handwritten code in several of the transformation benchmarks, but it performs poorly in the query and monadic benchmarks. In the query benchmarks, it is hampered by its use of a state monad to encode query traversals.

The performance of the Template Haskell based systems, TYB and Geniplate, stands in contrast to the other systems. With few exceptions, they consistently perform on par with (or in some cases, even better than) handwritten code.

5.4 Performance factors

Traversal strategy Different generic-programming systems use different strategies to perform traversals. Systems that use less efficient traversal strategies are at a disadvantage in these benchmarks regardless of the efficiency of their generic-programming primitives. This makes fair comparisons between different systems tricky and introduces confounding factors that must be considered when evaluating these benchmark results.

For many traversals, threading an accumulator through the computation is much more efficient than simply returning a result. Despite this, SYB, Alloy, EMGM and Smash do not use accumulators in their standard traversals. This hurts their performance in the query benchmarks (*sum*, *countName*, *listName* and *selectInt*). When applicable, both Geniplate and Uniplate automatically use accumulators and it is impossible not to use accumulators. In TYB, accumulator style is trivial to express and is used in the results in Figure 10 when applicable. To quantify this effect, Figure 11a compares versions of the query benchmarks for TYB that use accumulators against versions that do not.

Selective traversal Another confounding factor is that some systems (such as Alloy, Uniplate, and Geniplate) automatically skip parts of a traversal when a value is reached that has a type that cannot contain any of the target types. For example, a `String` cannot contain an `HsName` and so those parts of the traversal can be safely skipped.

Other systems, such as SYB, can skip parts of the traversal but must be explicitly told which types to skip. Explicitly listing all types that cannot contain a particular type is a heavy burden on the user when, as in the AST manipulation benchmarks, the number of types involved is large. Thus in the results in Figure 10, we have used the default behavior for each system with regards to skipping.

To quantify this effect, Figure 11b compares versions of the AST benchmarks for TYB that skip unnecessary parts of the traversal against versions that do not. The results for TYB in Figure 10 skip parts of a traversal when applicable.

Useless computations For all of the generic programming systems, `renumberInt` is significantly slower than the handwritten version. Upon investigation, we found that for TYB the performance difference is due to an extra bind (`>>=`) and `return` at the parts of the traversal without type-specific behaviour. For example, `mkM 'incInt` returns `return` on non-`Int` types. The surrounding `everywhereM` uses a monadic bind to thread traversal results through that `return`. Of course, these have no semantic effect on the results and should be optimized away, but with GHC 7.0.3 they are not. With GHC 7.4.1, they are optimized and the performance of TYB matches handwritten code. Some of the frameworks require non-trivial changes to port to GHC 7.4.1, so the numbers we report in Figure 10 are all based on 7.0.3.

Note that Geniplate does not speed up to match handwritten code when compiled with GHC 7.4.1. One possible cause is gratuitous monadic lifting of both construtors and values prior to monadic application (i.e., `ap`) similar to the inefficiencies that arise with `thfoldl` as discussed in section 4.2.

Finally, neither Uniplate nor Geniplate have a direct mechanism for query traversals. Instead they provide a mechanism for listing all objects of a particular type within an object. The user must then

fold over this list to construct a query result. This intermediate list costs performance on the query benchmarks.

6. Related Work

There are many generic-programming systems for Haskell, and even projects to extend Haskell to directly support generic programming [Jansson and Jeuring 1997; Hinze et al. 2002]. Hinze et al. [2007] and Rodriguez et al. [2008] survey several of these systems. We review only a few of them here and limit ourselves to systems with particularly interesting performance properties.

6.1 Uniplate

Uniplate [Mitchell and Runciman 2007] has two particularly notable aspects. The first is that Mitchell and Runciman include a performance evaluation of their system. The benchmarking results reported in that paper show the "direct" variant of Uniplate taking between 1.16 and 3.28 times the time of a handwritten traversal. On the other hand, our initial benchmark results showed Uniplate running between 6 and 10 times slower. When we reported this discrepancy to the authors of Uniplate, they determined that this was caused by certain intermediate structures not being eliminated as expected. We suspect this is due to differences in the optimizations implemented by different versions of the compiler. This is consistent with the great variation in the performance of generic-programming systems between compiler versions that was observed by Rodriguez et al. [2008]. An improved version of Uniplate is being developed that address some of these performance regressions, but it has not yet been publicly released. One of the advantages of TYB is that it does not require extensive tweaks or optimization pragmas to achieve its performance. Since it generates code similar to handwritten code, it has performance similar to that of handwritten code. This is a robust property that is independent of compiler version.

The second notable aspect of Uniplate is that it is based on the observation that "most traversals have value-specific behavior for just one type" [Mitchell and Runciman 2007]. For example, the non-generic parts of *selectInt* and *listName* are limited to the `Int` and `HsName` types respectively. By limiting the scope to such scenarios, the interface is greatly simplified.

Nevertheless, there are many times when the non-generic parts of the traversal involve multiple types. For example, an identifier-freshening pass in a compiler needs to deal with every binding form of an AST. In a language that distinguishes between declarations, expressions and statements, there may be binding forms in each of these different types. Thus while it may be worth making the single-type scenarios easy to write, in TYB we have chosen to keep the interface general enough to express general traversals.

6.2 Geniplate

Geniplate [Augustsson 2011] uses an interface similar to that of Uniplate, but uses Template Haskell to generate custom traversals. Thus, for the sorts of traversals that it supports, it executes efficiently. At present, there is no published information or literature about Geniplate other than the code itself.

Like Uniplate, it automatically skips parts of the traversal that contain no interesting types. The performance benefits of this are clear from the results in Figure 11b.

Also like Uniplate, Geniplate cannot express certain traversals involving multiple types. Operations that take only a single traversal in TYB might need multiple traversals or might not even be expressible when using Geniplate. For example, the identifier-freshening pass mentioned earlier that cannot be implemented in Uniplate cannot be implemented in Geniplate either. We found many examples of such traversals when implementing the Habit compiler.

Though they were developed independently without knowledge of each other, both Geniplate and TYB are similar in that they demonstrate the performance benefits of using Template Haskell to implement generic-programming systems. Geniplate does this for a Uniplate-style interface while TYB does this for the more general and flexible SYB-style interface. Note that the generality of TYB means it is easy to implement a Geniplate-like library in terms of TYB but the converse is not true.

6.3 Instant Generics

Instant Generics [Chakravarty et al. 2009; van Noort et al. 2008] uses a generic representation to uniformly represent the children of any given datatype in terms of sum and product types. A traversal can then be written in terms of that representation. Associated types are used to flexibly express the types in the generic representation.

Instant Generics requires the user to write a type class for each generic operation and to instantiate that class at appropriate types. The boilerplate parts of the traversal are handled by a default instance written by the user.

Instant Generics averages four times slower than the hand-written code, placing it behind TYB and Geniplate but notably faster than most other generic-programming systems.

6.4 Alloy

Alloy [Brown and Sampson 2009] was developed due to a need to have a high-performance generic-programming system. Its authors developed it while implementing a nanopass-based compiler [Sarkar et al. 2004] and developed it in order for their "compiler passes to traverse the abstract syntax tree quickly." Though we have chosen a different approach, our system is motivated by similar concerns.

Brown and Sampson include a sophisticated statistical analysis of the performance of their system and show it is faster than "existing approaches for traversing heterogeneously-typed trees."

Alloy does not directly feature a facility for constructing query traversals. Instead, a state or writer monad must be used to collect query results. This causes poor performance in the *sum*, *selectInt*, *countName* and *listName* benchmarks.

Alloy avoids traversing types that do not contain any types of interest. It does this automatically as a consequence of the structure of its design, whereas most other systems add this feature after the fact. In the *countName* and *listName* benchmarks, implementing this idea in TYB cuts the runtime in half.

Alloy requires a large number of class instances for the datatypes being traversed. Alloy includes a tool for generating these instances, so this is not a burden on the programmer. Nevertheless, the source code for these instances can be quite large. For example, with the AST types in `Language.Haskell.Syntax` the generated *source* code for these instances is 1.3 MB when used with overlapping and 3.8 MB when used without overlapping. Compile times on these files can be quite long. On larger types such as the AST types in `Language.Haskell.Exts`, compilation failed to complete on our test machine even after several hours.

6.5 Prototyping Generic Programming in Template Haskell

Norell and Jansson [2004] discuss the use of Template Haskell for prototyping generic-programming systems. In particular, they present prototype implementations of PolyP and Generic Haskell written in Template Haskell and use these prototypes to relate and contrast PolyP and Generic Haskell with each other. They do not consider the question of performance.

6.6 Optimizing generics is easy!

Magalhães et al. [2010] demonstrate that, with appropriate tweaking of various compiler inlining flags, many generic-programming

traversals can be optimized to more closely match the performance of handwritten code. Their approach is limited by how much control the compiler provides over the inlining process, and its effectiveness varies significantly depending on the design of the underlying generic-programming system.

In contrast, TYB does not need aggressive inlining in order to be efficient. By generating specialized traversals, it already does the work that inlining would do.

7. Conclusion

The idea of generating traversal code via metaprogramming is not new and is a widely used technique in other languages. However, in the Haskell community it has largely been eschewed in favor of type and class based techniques. Nevertheless, the metaprogramming approach offers significant performance advantages.

Since TYB generates code at compile time, it does not pay the overheads seen in most other generic-programming systems for Haskell. It generates code that is similar to what a programmer would write by hand, and thus it runs as fast as handwritten code. But unlike handwritten code, it is easy to change the design of the traversal and to experiment with different approaches to see which performs best. For example, with TYB, it requires changing only a few lines to move from a list representation to a set representation for query results or from a non-accumulating recursion to an accumulating recursion.

TYB has been used by our group to implement over a dozen passes in the Habit compiler where it has proven both useful and effective. The ease with which new traversals can be written encourages factoring compiler passes into small well defined traversals instead of combining multiple operations into one large pass. At the same time, TYB does not incur the overheads seen in other generic-programming systems, and we can rely on the efficiency of the resulting traversals.

Acknowledgments

Our thanks go to those who have helped improve this paper. Neil Mitchell helped us ensure that our benchmarking of his system was accurate. Feedback from Mark P. Jones, Andrew Tolmach, J. Garrett Morris, Simon Peyton Jones and the anonymous reviewers helped improve the presentation of this paper.

References

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

Alan Bawden. Quasiquotation in Lisp. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, volume NS-99-1 of *BRICS Notes Series*, pages 4–12, Aarhus, Denmark, January 1999. BRICS.

Neil C. C. Brown and Adam T. Sampson. Alloy: fast generic transformations for Haskell. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, Haskell '09, pages 105–116, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-508-6. doi: 10.1145/1596638.1596652.

Manuel M. T. Chakravarty, Gabriel C. Ditu, and Roman Leshchinskiy. Instant generics: Fast and easy, 2009.

HASP Project. The habit programming language: The revised preliminary report, November 2010. URL `http://hasp.cs.pdx.edu/habit-report-Nov2010.pdf`.

Ralf Hinze, Johan Jeuring, and Andres Löh. Type-indexed data types. In Eerke Boiten and Bernhard Möller, editors, *Mathematics of Program Construction*, volume 2386 of *Lecture Notes in Computer Science*, pages 77–91. Springer Berlin / Heidelberg, 2002. ISBN 978-3-540-43857-1. doi: 10.1007/3-540-45442-X_10.

Ralf Hinze, Johan Jeuring, and Andres Löh. Comparing approaches to generic programming in Haskell. In Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring, editors, *Datatype-Generic Programming*, volume 4719 of *Lecture Notes in Computer Science*, pages 72–149. Springer Berlin / Heidelberg, 2007. ISBN 978-3-540-76785-5. doi: 10.1007/978-3-540-76786-2_2.

Patrik Jansson and Johan Jeuring. Polyp—a polytypic programming language extension. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '97, pages 470–482, New York, NY, USA, 1997. ACM. ISBN 0-89791-853-3. doi: 10.1145/263699.263763.

Oleg Kiselyov. Smash your boiler-plate without class and typeable, August 2006. URL `http://article.gmane.org/gmane.comp.lang.haskell.general/14086`.

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

José Pedro Magalhães, Stefan Holdermans, Johan Jeuring, and Andres Löh. Optimizing generics is easy! In *Proceedings of the 2010 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, PEPM '10, pages 33–42, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-727-1. doi: 10.1145/1706356.1706366.

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

Ulf Norell and Patrik Jansson. Prototyping generic programming in template haskell. In Dexter Kozen, editor, *Mathematics of Program Construction*, volume 3125 of *Lecture Notes in Computer Science*, pages 314–333. Springer Berlin / Heidelberg, 2004. ISBN 978-3-540-22380-1. doi: 10.1007/978-3-540-27764-4_17.

Bruno C. d. S. Oliveira, Ralf Hinze, and Andres Löh. Extensible and modular generics for the masses. In *Trends in Functional Programming*, volume 7 of *Trends in Functional Programming*, pages 199–216. Intellect, 2006. ISBN 978-1-84150-188-8.

Bryan O'Sullivan. Criterion version 0.6.0.1, January 2012. URL `http://hackage.haskell.org/package/criterion/`.

Alexey Rodriguez, Johan Jeuring, Patrik Jansson, Alex Gerdes, Oleg Kiselyov, and Bruno C. d. S. Oliveira. Comparing libraries for generic programming in Haskell. In *Proceedings of the first ACM SIGPLAN symposium on Haskell*, Haskell '08, pages 111–122, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-064-7. doi: 10.1145/1411286.1411301.

Dipanwita Sarkar, Oscar Waddell, and R. Kent Dybvig. A nanopass infrastructure for compiler education. In *Proceedings of the ninth ACM SIGPLAN international conference on Functional programming*, ICFP '04, pages 201–212, New York, NY, USA, 2004. ACM. ISBN 1-58113-905-5. doi: 10.1145/1016850.1016878.

Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, Haskell '02, pages 1–16, New York, NY, USA, 2002. ACM. ISBN 1-58113-605-6. doi: 10.1145/581690.581691.

Thomas van Noort, Alexey Rodriguez, Stefan Holdermans, Johan Jeuring, and Bastiaan Heeren. A lightweight approach to datatype-generic rewriting. In *Proceedings of the ACM SIGPLAN workshop on Generic programming*, WGP '08, pages 13–24, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-060-9. doi: 10.1145/1411318.1411321.