Optimizing Generics Is Easy!

José Pedro Magalhães

Stefan Holdermans

Johan Jeuring

Andres Löh

Technical Report UU-CS-2009-022
October 2009

Department of Information and Computing Sciences
Utrecht University, Utrecht, The Netherlands
www.cs.uu.nl

ISSN: 0924-3275

Department of Information and Computing Sciences
Utrecht University
P.O. Box 80.089
3508 TB Utrecht
The Netherlands

Optimizing Generics Is Easy!

José Pedro Magalhães[1] Stefan Holdermans[1] Johan Jeuring[1,2] Andres Löh[1]

[1]Department of Information and Computing Sciences, Utrecht University, P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
[2]School of Computer Science, Open University of the Netherlands, P.O. Box 2960, 6401 DL Heerlen, The Netherlands

{jpm,stefan,johanj,andres}@cs.uu.nl

Abstract

Datatype-generic programming increases program reliability by reducing code duplication and enhancing reusability and modularity. Several generic programming libraries for Haskell have been developed in the past few years. These libraries have been compared in detail with respect to expressiveness, extensibility, typing issues, etc., but performance comparisons have been brief, limited, and preliminary. It is widely believed that generic programs run slower than hand-written code. In this paper we present an extensive benchmarking suite for generic functions and analyze the potential for automatic code optimization at compilation time. Our benchmark confirms that generic programs, when compiled with the standard optimization flags of the Glasgow Haskell Compiler, are substantially slower than their hand-written counterparts. However, we also find that more advanced optimization capabilities of GHC can be used to further optimize generic functions, sometimes achieving the same efficiency as hand-written code.

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

General Terms Languages

Keywords benchmarking, functional programming, generic programming, Haskell, optimization

1. Introduction

Datatype-generic programming, as defined by Gibbons (2007), has been around for many years (Backhouse et al. 1999). Its prominence is especially noted in the lazy, strongly-typed functional language Haskell (Peyton Jones et al. 2003), where at least twelve different approaches have appeared. Rodriguez Yakushev et al. (2009) presented a detailed comparison of nine libraries for generic programming (three new libraries have appeared since then). This comparison contains a brief performance analysis. This analysis indicates that the use of a generic approach could result in an increase of the running time by a factor of as much as 80. Van Noort et al. (2008) also report severe performance degradation when comparing a generic approach to a similar but type-specific variant. While this is typically not a problem for smaller examples, it can severely impair adoption of generic programming in larger contexts. This problem is particularly relevant because generic programming techniques are especially applicable to large applications where performance is crucial, such as structural editors or compilers.

To understand the source of performance degradation when using a generic function from a particular generic programming library, we have to analyse the implementation of the library. The fundamental idea behind generic programming is to represent all types and values by a small set of datatypes. Types in this set are called the generic representation types, as they are the building blocks of the structural representation of all other types. Haskell supports the definition of algebraic datatypes, which can be structurally represented as sums of products. This means that if we have a way to represent sums (alternatives) and products (tuples), we can represent all algebraic datatypes. Functions can then be defined to operate on this sum-of-products structure. When coupled with conversion functions from the original datatype into the sum-of-products structure and back, these functions effectively become datatype-generic, in the sense that they will work for all datatypes.

While the conversion functions are typically trivial and can be automatically generated, the overhead they impose is not automatically removed. Conversions to and from the generic representations, as well as value-level representations of types, are not eliminated by compilation, and conversions are performed at run-time. In general, conversions are the source of inefficiency for generic programming libraries. In the past, approaches of generic programming were not implemented as a library, but instead as code generators or preprocessors (Hinze et al. 2007). This meant that optimizations (such as automatic generation of type-specialized variants of generic functions) could be implemented externally. With the increase in expressivity of the Haskell type system, particularly with the extensions of the flagship compiler GHC, library-based approaches have become more popular, as they are able to offer the same functionality without the need to install an external application. However, this also means that all optimizations have to be performed by the compiler, as the library approach no longer generates code itself.

GHC compiles by transformation, first converting the input into a core language and then proceeding to transform the core into more optimized versions, in a series of sequential passes. While it performs a wide range of optimizations, by default it seems to be unable to remove the overhead incurred by using generic representations. Therefore generic libraries perform slower than hand-written type-specific counterparts. Alimarine and Smetsers (2004) show that in many cases it is possible to remove all overhead from generics by performing a specific form of symbolic evaluation. In fact, their approach is not restricted to optimizing generics, and GHC naturally performs symbolic evaluation as part of its optimizations. We observe that GHC can already optimize generic functions, in many cases achieving the same performance as hand-written code, without the need for any additional manipulation of the compiler.

The remainder of this paper discusses how GHC can optimize generic functions, and tests the optimizations through benchmarking. Specifically, our contributions are the following:

- Highlighting the importance of inlining for optimization of generic programs, by inspecting the generated core code with different levels of keenness to inline.
- Realizing that GHC already provides mechanisms for optimizing generic programs up to the efficiency of hand-written code. This also emphasizes the importance of being able to customize the behavior of the inliner.
- An overview of some important aspects of the design of a benchmarking suite for functional programs, which are crucial for obtaining relevant and meaningful results.
- The results of benchmarking several generic functions, applied to different datatypes and compiled with different optimization levels. We analyze the results and gather a significant amount of information and evidence to drive further optimization of current generic programming libraries.

We proceed by introducing a simple but representative library for generic programming in Haskell (Section 2). In Section 3 we follow the evolution of some generic code through the GHC simplifier to understand the source of inefficiencies. Section 4 discusses some important details of benchmarking functional programs, which we use in our benchmark in Section 5. In Section 6 we conclude, point to directions for future work and give instructions for authors and users of generic programming libraries interested in optimizing the performance of their programs.

2. The `regular` library

The `regular` library for generic programming[1] is used in the rewriting framework of Van Noort et al. (2008), and can be viewed as a simple version of `multirec` (Rodriguez Yakushev et al. 2009). While `multirec` uses higher-order fixed-points for representing mutually recursive datatypes, `regular` restricts itself to single, regular datatypes. We use it as a starting point for our optimization efforts because it is both representative for a generic programming library and relatively simple. It is representative in the sense that it employs a sum-of-products view with fixed-points. Both `multirec` and `emgm` (Oliveira et al. 2007) use a sum-of-products view (although they treat fixed-points differently). While `regular` uses a significant extension of Haskell 98 (type families), it remains simple in the sense that its inner workings are relatively straightforward. Therefore we also expect generated core code from `regular` to be easier to understand than, say, code from `syb` (Lämmel and Peyton Jones 2003). We discuss `emgm`, `multirec` and `syb` in our benchmark (Section 5.3), but for now we focus on `regular`.

2.1 Generic representation

The core of a generic programming library is the representation of datatypes. In `regular`, type families are used to associate a pattern functor with each datatype. This pattern functor expresses a sum-of-products fixed-point view on the datatype. For this, a number of primitive generic representation types are required:

```
data (f + g) r = L (f r) | R (g r)
data (f × g) r = f r × g r
data K a     r = K a
data I       r = I r
data U       r = U
```

[1] `http://www.cs.uu.nl/wiki/GenericProgramming/Regular`

```
infixr 6 +
infixr 7 × .
```

Different alternatives (constructors) of a datatype are encoded with the $+$ type. Constructors with no arguments are encoded with the unit type *U*, and multiple arguments are represented with the product type $\times$. The sum and product types are declared as right-associative infix type constructors.

The arguments of a constructor are either constant types (represented with *K*), or a recursive occurrence of the type being defined (represented with *I*). All representation types are functorial, carrying the type argument *r* which denotes the recursive type, which is used in *I*.

Generic functions such as *show* or *read* also need information about constructor names and fixities. These are encoded with the *C* representation type and the *Constructor* class:

```
data C c f r = C (f r)
class Constructor c where
  conName   :: t c (f :: ∗ → ∗) r → String
  conFixity :: t c (f :: ∗ → ∗) r → Fixity
  conFixity = const Prefix .
```

This style of encoding constructors, which is also used in `multirec`, requires the declaration of an auxiliary datatype per constructor of the datatype to represent. We give an example below.

We encapsulate the representation of a datatype and conversion functions between a datatype and its representation in the *Regular* type class:

```
class Regular a where
  type PF a :: ∗ → ∗
  from      :: a        → PF a a
  to        :: PF a a → a .
```

To use a generic function on some datatype *a*, an instance of *Regular a* has to be provided. The type family *PF* is used to represent the pattern functor of a type: its view as a fixed-point of a sum-of-products. The *from* and *to* functions witness the isomorphism between *a* and *PF a a*. Note that *PF a*, of kind $* \to *$, denotes the generic representation type of *a*. The last argument of *PF* denotes the type of the recursive occurrences. Since we use *PF a a*, this means that we have a shallow generic encoding: the recursive occurrences are again values of the original type *a*.

To better understand the generic representation in `regular`, we give the instance of *Regular* for a datatype representing logic expressions:

```
data Logic = Logic ∨ Logic    -- disjunction
           | Var String       -- variables
           | Not Logic        -- negation
           | Const Bool       -- true and false .
```

For the constructor representations, we create a datatype per constructor and give an instance of the *Constructor* class:

```
data (∨)
data Var
data Not
data Const
instance Constructor (∨) where
  conName _ = ":||:"
  conFixity _ = Infix RightAssociative 6
instance Constructor Var   where conName _ = "Var"
instance Constructor Not   where conName _ = "Not"
instance Constructor Const where conName _ = "Const" .
```

The instance of *Regular* exposes the fixed-point view of *Logic* and conversion functions:

```
instance Regular Logic where
  type PF Logic = C (∨)     (I × I)
                + C Var     (K String)
                + C Not     I
                + C Const   (K Bool)
  from (p ∨ q)    = L           (C ((I p) × (I q)))
  from (Var x)    = R (L        (C (K x)))
  from (Not p)    = R (R (L (C (I p))))
  from (Const b)  = R (R (R (C (K b))))
  to (L         (C ((I p) × (I q)))) = p ∨ q
  to (R (L      (C (K x))))          = Var x
  to (R (R (L (C (I p)))))           = Not p
  to (R (R (R (C (K b)))))           = Const b .
```

Note the shallow encoding in the recursive cases: *Not p*, for instance, is converted to *R (R (L (C (I p))))*. The *p* variable is of type *Logic*. This is therefore a one-level generic structure, with the original datatype occurring at the recursive positions.

All this code is necessary for each datatype intended to be used with `regular`. Fortunately, its structure is easily derivable from the datatype declaration, and the library comes with Template Haskell (Sheard and Peyton Jones 2002) code to automatically generate it.

2.2 Generic functions

We proceed to define generic functions to operate on regular datatypes. In `regular`, generic functions are defined using type classes. Let us start with generic map. Since we have a fixed-point view on data, our representation types are functors, so this function is trivial to write. We start by defining a type class:

```
class GMap f where
  gmap :: (a → b) → f a → f b .
```

We then write an instance for each of the generic representation types. The only interesting case for generic map is *I*, where the function to be mapped is applied. Units and constants are returned unchanged. For constructors, sums and products, we keep the structure and recursively call the function on the children:

```
instance GMap I where
  gmap f  (I r)    = I (f r)
instance GMap (K a) where
  gmap _ (K x)     = K x
instance GMap U where
  gmap _ U         = U
instance (GMap f, GMap g) ⇒ GMap (f + g) where
  gmap f  (L x)    = L (gmap f x)
  gmap f  (R x)    = R (gmap f x)
instance (GMap f, GMap g) ⇒ GMap (f × g) where
  gmap f  (x × y) = gmap f x × gmap f y
instance (GMap f) ⇒ GMap (C c f) where
  gmap f  (C x)    = C (gmap f x) .
```

We analyze the generated code for this function in Section 3.1.

Making use of the constructor information, we can write generic show. For simplicity we present an inefficient version using (++). A version returning *ShowS* is written similarly.

```
class GShow f where
  gshowf :: (a → String) → f a → String .
```

Given the shallow encoding of `regular`, we need to pass a function to apply at the recursive points. This is the first argument of *gshowf*. This function is applied in the *I* case. For constants, we use the standard Haskell *show* from the *Show* class. At the constructor, we print the constructor name as defined in the *Constructor* instance.

```
instance GShow I where
  gshowf f  (I r)     = f r
instance (Show a) ⇒ GShow (K a) where
  gshowf _ (K x)      = show x
instance GShow U where
  gshowf _ U          = ""
instance (GShow f, GShow g) ⇒ GShow (f + g) where
  gshowf f  (L x)     = gshowf f x
  gshowf f  (R x)     = gshowf f x
instance (GShow f, GShow g) ⇒ GShow (f × g) where
  gshowf f  (x × y)   = gshowf f x ++ "  " ++ gshowf f y
instance (Constructor c, GShow f) ⇒ GShow (C c f) where
  gshowf f  cx@(C x) = "(" ++ conName cx ++ "  "
                        ++ gshowf f x ++ ")" .
```

We can now write a top-level function that calls *gshowf* with the right arguments:

```
gshow :: (Regular a, GShow (PF a)) ⇒ a → String
gshow x = gshowf gshow (from x) .
```

Given any value of a datatype of which there is a *Regular* instance, we can convert it to its structural type with *from* and invoke *gshowf*. The recursive points are again of type *a*, so they are shown using *gshow*, which will take care of the conversion to the structural type. This type of indirect recursion is present in most generic functions in `regular`.

Many other functions can be written in `regular`, such as equality, value generation and folds. These functions are actually all part of the library.

3. Optimizing generics through inlining

Previous research has shown that generic functions perform worse than hand-written equivalents, sometimes taking up to 80 times longer to produce the same result (Rodriguez Yakushev 2009, Figure 4.9). While many applications might not be affected by this performance overhead, it can completely preclude generics in some cases. In at least one case, this spurred the development of a new generic programming library (Brown and Sampson 2009).

The source of inefficiency in generic programs is the overhead of specialization. The generated code for generic functions will typically contain explicit conversions to and from the generic representation types, since the function's behavior was specified on these types. However, a more clever specialization could eliminate all of the generic representation. Indeed Alimarine and Smetsers (2004) have shown that generic functions can benefit from symbolic evaluation, enabling the generation of code which contains no constructors others than those of the concrete type the function operates on.

We argue that an extra step of symbolic evaluation is not necessary, since clever inlining coupled with the standard optimizations of GHC can perform the same task. The inliner of GHC provides several flags to tweak its behavior, as well as supporting `INLINE` pragmas to annotate functions (Peyton Jones and Marlow 2002). However, even its authors admit that "inlining is a black art"; careless use of the inliner can easily cause code bloat or even decrease performance. In this section we aim to illustrate the results of inlining in the compilation of two example generic functions: identity and show.

3.1 Generic identity

We define generic identity using the *gmap* function defined in Section 2.2:

$$gid :: (Regular\ a, GMap\ (PF\ a)) \Rightarrow a \to a$$
$$gid = to \circ gmap\ id \circ from\ .$$

This "one-layer identity" simply applies the identity function to the children of its input. It is clear that for a properly written instance of the *Regular* class, $gid \equiv id$. To evaluate the result of the optimizations on the generated code for this function, we use a specialized version instead:

$$gid_{Logic} :: Logic \to Logic$$
$$gid_{Logic} = to \circ gmap\ id \circ from\ .$$

This is the typical use-case: a user applies a generic function to a concrete type. Additionally, the compiler now knows which *Regular* instance to use. With this information, it is trivial to see that $gid_{Logic}\ l \equiv l$ forall l of type *Logic*:

$$\begin{array}{ll}
 & to\ (gmap\ id\ (from\ l)) \\
\Rightarrow & \{\text{ choose } l \text{ to be } p \vee q \text{ (other constructors are similar) }\} \\
 & to\ (gmap\ id\ (from\ (p \vee q))) \\
\equiv & \{\text{ definition of } from_{Logic}\ \} \\
 & to\ (gmap\ id\ (L\ (C\ (I\ p \times I\ q)))) \\
\equiv & \{\text{ definition of } gmap_{+}, gmap_{C}, gmap_{\times}\ \} \\
 & to\ (L\ (C\ (gmap\ id\ (I\ p) \times gmap\ id\ (I\ q)))) \\
\equiv & \{\text{ definition of } gmap_{I}\ \} \\
 & to\ (L\ (C\ (I\ (id\ p) \times (I\ (id\ q))))) \\
\equiv & \{\text{ definition of } id, to_{Logic}\ \} \\
 & p \vee q\ .
\end{array}$$

But can GHC reach a similar conclusion through its optimization phases? We were positively surprised to see that compiling gid_{Logic} with the flag `O1` alone produces the following core code[2]:

$$gid^{O1}_{Logic} :: Logic \to Logic$$
$$gid^{O1}_{Logic} = \lambda(x :: Logic) \to to\ (from\ x)\ .$$

The generic function disappeared completely, and only the conversion to and from the generic representation remains. However, this is still insufficient, since we also know that $to_{Logic} \circ from_{Logic} \equiv id$. Compiling with `O2` produces the same code for gid_{Logic}.

At this point, we could use rewrite rules (Peyton Jones et al. 2001) to encode this last equality. However, there is no need to manually specify such rules. GHC offers some flags to tweak the behavior of the inliner, which we can use to achieve a similar result. We summarize them with their defaults values in Table 1.

Flag	Default	Abbr.
`-funfolding-creation-threshold`	45	CT
`-funfolding-fun-discount`	6	FD
`-funfolding-keeness-factor`	1.5	KF
`-funfolding-use-threshold`	6	UT

Table 1. GHC flags to tweak the inliner.

The behavior of each flag is the following.

`-funfolding-creation-threshold` governs the exposure of a function's code to the interface file. GHC, as a modular compiler, compiles each module separately and generates an interface file, which is then used when the module is imported. To allow cross-module inlining, some functions have their entire code exposed in the interface file. However, larger functions might not be exposed by default, since this would create a large interface file. Increasing `-funfolding-creation-threshold` raises the limit for exposing the definition of a function in the interface file.

`-funfolding-use-threshold` defines the threshold for actually performing inlining. Below this size, a function definition will be unfolded at a call site. If the function size is higher than this threshold, it will not be unfolded. This is generally the most critical value in determining inlining.

`-funfolding-fun-discount` affects the calculation of the size of a function, which in turn affects the decision to inline it or not.

`-funfolding-keeness-factor` is multiplied by any discounts that apply to a function when deciding to inline. Higher values increase the keenness to inline a function.

All these parameters either depend on or affect the calculation of the size of a function. More details about each of these factors and the calculation of the size of a function are provided by Peyton Jones and Marlow (2002), or directly in the GHC source code (namely the *CoreUnfold* module). The abbreviations are used in function naming: $gshow^{CT90UT20}_{Logic}$, for instance, represents the function *gshow* specialized to the *Logic* datatype, compiled with `-funfolding-creation-threshold=90` and also `-funfolding-use-threshold=20`. In this section, whenever we use unfolding flags we always use optimization level `O2`, so this is not mentioned in the name.

Returning to our gid_{Logic} example, it turns out that compiling with `-O2 -funfolding-use-threshold=60` produces the following code:

$$gid^{O2UT60}_{Logic} :: Logic \to Logic$$
$$gid^{O2UT60}_{Logic} = \lambda(x :: Logic) \to x\ .$$

This means GHC can specialize gid_{Logic} to nothing more than the identity function, when given enough time and incentive to perform inlining.

Note, however, that extra inlining is not always desirable. The total amount of generated core code for this test increased by 72%. This is because inlining a function that is used more than once will cause code duplication. A sample binary using this function, however, increased only 2% in size, which is acceptable. But code duplication is not the only negative side-effect of inlining. Work duplication can also occur, if, for instance, we inline x in the expression $x + x$. This results in increased running time, which is something our benchmark can detect (Section 5).

3.2 Generic show

The generic shallow identity function is not a representative test for a generic programming library. We now turn our attention to *gshow* as a more realistic test case. As previously, we start by defining a specialized version:

$$gshow_{Logic} :: Logic \to String$$
$$gshow_{Logic} = gshow\ .$$

Using "standard" optimization flags only, we find that `O1` creates a version of *gshow* specialized to *Logic*, but still performing runtime type comparison and using the generic representation types:

$$\begin{array}{l}
gshow^{O1}_{Logic} :: Logic \to String \\
gshow^{O1}_{Logic} = \lambda(x :: Logic) \to \\
\quad \textbf{case}\ (from\ x)\ \text{`}cast\text{`}\ (sym\ (trans \ldots))\ \textbf{of}\ w\ \{ \\
\qquad L\ y \to \ldots \\
\qquad R\ y \to \ldots \\
\quad \}\ .
\end{array}$$

[2] Core language is a variant of System F where all variables are explicitly typed.

(Note that in core code the **case** statement has a case binder as an extra feature. It binds *w* to the value of the scrutinee, but it is not used in our examples. The *cast* is necessary due to core being a typed language, but these are erased at a later stage.)

Optimization level `O2` does not produce a significant improvement of this code. If we start tweaking the unfolding settings, we need to increase both the creation and the use threshold to notice an improvement in the generated code:

$$
\begin{array}{l}
gshow^{\texttt{CT90UT20}}_{Logic} :: Logic \rightarrow String \\
gshow^{\texttt{CT90UT20}}_{Logic} = \lambda(x :: Logic) \rightarrow f\,@Logic\ gshow^{\texttt{CT90UT20}}_{Logic} \\
\quad (\textbf{case}\ x\ \textbf{of}\ w\ \{ (\vee)\ p\ q \rightarrow L@(C \vee (I \times I)) \\
\qquad\qquad\qquad\qquad\qquad\qquad @(C\ Var\ (K\ [Char]) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad + (C\ Not\ I + C \ldots)) \\
\qquad\qquad\qquad \ldots\}) \,.
\end{array}
$$

In $gshow^{\texttt{CT90UT20}}_{Logic}$ we notice that the type coercions are gone, but we are still left with a two-step process: first decompose *Logic* constructors into their generic representation, and then consume those (function *f*, which we omit) to produce the result *String*. Note that, in core code, type application is made explicit with the @ operator (as in *value*@*Type*), and there is no syntactic sugar for infix operators.

By increasing the use threshold further, we achieve a specialized version which no longer uses any generic representation types. The code is very similar to that generated for a hand-written variant of *show*:

$$
\begin{array}{l}
gshow^{\texttt{CT90UT30}}_{Logic} :: Logic \rightarrow String \\
gshow^{\texttt{CT90UT30}}_{Logic} = \lambda(x :: Logic) \rightarrow \textbf{case}\ x\ \textbf{of}\ w\ \{ \\
\quad (\vee)\ p\ q \rightarrow (+\!\!+) \ldots\ gshow^{\texttt{CT90UT30}}_{Logic}\ p\ \ldots\ gshow^{\texttt{CT90UT30}}_{Logic}\ q\ \ldots \\
\quad Var\ v \quad\ \rightarrow (+\!\!+) \ldots\ show\ v\ \ldots \\
\quad Not\ p \quad \rightarrow (+\!\!+) \ldots\ gshow^{\texttt{CT90UT30}}_{Logic}\ p\ \ldots \\
\quad Const\ b \rightarrow (+\!\!+) \ldots\ show\ b\ \ldots\} \,.
\end{array}
$$

Optimizing generics is easy! With the right amount of tweaking in the unfolding settings, GHC can generate type-specific functions from a generic function definition. These specialized functions contain no generic representation overhead, and are similar to the code that is generated for hand-written type-specific functions.

Unfortunately, tweaking the unfolding options is a rather drastic approach, since it will affect all generated code, and not only the generic functions we want to specialize. We have also tried using other, more localized techniques to achieve the same effect. GHC also provides the `INLINE` pragma[3], which serves to annotates a function that the programmer wants to have inlined. However, this serves only to reduce the function's cost for the inlining calculation, and it proved not to be enough to achieve the same results. Either the pragma does not reduce the cost enough to trigger inlining, or other factors override the pragma. `SPECIALIZE` pragmas are also not enough for the optimization we want: they achieve the same specialization as compilation with `O1` or `O2` does, and hence do not remove all of the generic overhead.

4. Benchmarking functional programs

To determine the actual performance of generic functions with relation to their hand-written counterparts, we have developed a benchmarking suite of generic functions. Benchmarking is, in general, a complex task, and a lazy language imposes even more challenges on the design of the benchmark. We require the following functionality from a benchmarking framework:

Reporting CPU times A benchmark can typically compute a variety of measures, such as user CPU time, system CPU time, or real (wall clock) time. Other measures may be used: Peyton Jones and Marlow (2002), for instance, found a correlation between run-time and memory allocation and hence chose to benchmark the latter. For our benchmark, we chose to measure CPU time. Wall clock time is not relevant for the comparison of performance of single functions, since we do not want to measure how long a user waits for a result, but instead how long the CPU is busy with the computation. We do not measure user and system CPU time separately because there are hardly any system calls in the code we benchmark.

Repeatability We want to be able to run tests multiple times, so we can average over multiple runs and possibly analyze the standard deviation or outliers. Each run has to be independent from previous runs.

Independence from operating system Our benchmarking suite should be available for execution on different machines running different operating systems. This enables not only a comparison of the optimizations of the compiler through different environments but also the development of the suite on different operating systems.

Reliability We want to be certain that our tests are benchmarking the running time of a specific generic function. Functionality such as test data generation and result outputting might also be present and impact the running time of the test.

Different compilation flags Since we want to compare different optimization flags, it has to be easy to run the same tests compiled with different flags.

To fulfill these criteria, we designed a benchmarking suite where each test is compiled as an independent program. Each program consists of getting the current CPU time, running the test, getting the updated CPU time and outputting the difference of the times. The Unix `time` command performs a similar task, but unfortunately no equivalent is immediately available in Microsoft Windows. By including the timing code with the test (using the portable function *System.CPUTime.getCPUtime*) we work around this problem.

This also ensures easy repeatability without running into problems with laziness. A package such as `benchpress`[4] provides a way to benchmark an IO action a number of times, but this is only useful if most of the time of the action is spent in IO-related tasks. For our tests, all the time is spent on pure functions, with IO being used only to output the result. This means that only the first run is accurate, as all the subsequent runs use the cached result and simply output it again.

To ensure reliability of the benchmark, we use profiling, which gives us information about which computations most time is being spent on. For each of the tests, we ensure that at least 50% of the time is spent on the function we want to benchmark. Profiling also gives information about the time it takes to run a program, but we do not use those figures since they are affected by the profiling overhead. An approach similar to the `nofib` benchmark suite (Partain 1993) is not well-suited to our case, as our main point is not to compare an implementation across different compilers, but instead to compare different implementations on a single compiler.

A top-level script (also written in Haskell) takes care of compiling all the tests with the same flags, invoking them a given number of times, parsing and accumulating results as each test finishes, and calculating and displaying the average running time at the end, along with some system information.

[3] `http://www.haskell.org/ghc/docs/latest/html/users_guide/pragmas.html#inline-noinline-pragma`

[4] `http://hackage.haskell.org/package/benchpress`

5. A benchmarking suite for generics

To better understand the performance impact of the optimizations described in Section 3, and to provide a more comprehensive report on the performance of generic programs, we designed a benchmarking suite of generic programs. It comprises a number of typical and representative generic functions written on a few different generic libraries. We ran it with several different optimization flags, as well as across different operating systems.

5.1 Generic functions

We benchmarked five different generic functions.

eq Equality is a well-known generic function, and its use is nearly universal. It is a generic consumer, in the sense that the generic type is an argument. It takes two generic arguments (albeit of the same type) and it has to traverse them fully to determine their structural equality.

map Map, defined in Haskell 98 as the *fmap* method of the *Functor* class, applies a function to all elements of a container type. Not all types can be mapped over: only those of kind $* \rightarrow *$. Consequently, not all libraries can express *map* adequately, since it requires dealing with higher-kinded types.

show Transforming a value into a *String* is also a very common generic function. It is also a consumer, and is present in most generic programming libraries. It requires constructor information, such as name and fixity.

read The inverse of *show* is *read*. It is also a generic function, but typically harder to express than *show*. It is a producer, in the sense that the generic type is the result type. It also requires constructor information.

update There are several types of update functions that can be written. The common goal is to transform a value of a datatype without changing its type. In our benchmark, we choose to transform all odd *Int* values by adding one to them, or to prepend all non-empty *String* values with a `"y"`. These are merely representative transformations of what is done in practice.

5.2 Datatypes

Unlike Rodriguez Yakushev et al. (2008), we do not intend to evaluate datatype support across different libraries. Therefore we use only two datatypes.

data *Tree a* = *Bin a* (*Tree a*) (*Tree a*) | *Leaf*

The *Tree* datatype is a simple labeled binary leaf tree. The label type is a parameter. This allows us to define a *map* for *Tree*.

```
data Logic = Impl Logic Logic | Equiv Logic Logic
           | Conj Logic Logic | Disj  Logic Logic
           | Not Logic | Var String | T | F
```

The *Logic* type is very similar to the one we introduced in Section 2.1, only with more constructors. The intention is to test the performance of generic programming libraries on datatypes with many constructors. Since many libraries represent algebraic datatypes as a nested sum of products, we suspect this nesting can cause inefficiency. We also avoid using infix constructors, to simplify the task of generic *show* and *read*. From this point on, whenever we mention the *Logic* type we are referring to this updated variant. The same holds for the *Var* and *Not* constructors.

5.3 Generic libraries

We have chosen a few representative, mainstream and maintained libraries to benchmark.

`emgm` Extensible and Modular Generics for the Masses (Hinze 2006; Oliveira et al. 2007) now exists as a mature, large and well-maintained library[5]. Its fundamental characteristic is to encode datatype representations through a type class.

`syb` Scrap Your Boilerplate (Lämmel and Peyton Jones 2003, 2004) is a very popular library based on generic combinators and type-safe cast. It comes with the GHC compiler.

`regular` This is the library we describe in Section 2.

`multirec` This library was introduced by Rodriguez Yakushev et al. (2009), and is the first approach able to express mutually recursive datatypes, making it possible to define folds and zippers on such types. Structurally, it is very similar to `regular`, also using type families to represent the generic structure of datatypes. However, it makes use of a few more advanced concepts to deal with mutual recursion.

We have also implemented all the functions in a type-specific hand-written variant. We benchmark the generic functions' performance against this hand-written standard. We take optimization level `O1` as the standard, so we plot all the values relative to the hand-written code optimized at level `O1`. We use a logarithmic scale (base 2) and set the x-axis to cross the y-axis at 1. Results where the performance is better than hand-written code compiled with `O1` therefore show as bars below this crossing line.

5.4 Results

We present some results from our benchmark, along with a discussion of some interesting highlights.

5.4.1 Optimization flags

We present a series of results comparing the performance of the generic functions when compiled with different optimization flags. While we focus on the `UT` flag in the results presented, we have also mentioned the existence of the `KF` and `FD` flags (Table 1). We have benchmarked the tweaking of these flags, and have found that while they also increase performance in certain cases, their behavior is not better than that of `UT`, namely in providing better results or preventing negative performance impact. Therefore, we show only the results of tweaking `UT`. The `CT` flag falls under a different category, since it affects the insertion of functions to inline on the header files (to allow for cross-module inlining). We have used different levels of `CT` in all our tests with `KF` and `FD`, just like the ones we present for `UT`.

Equality In Figure 1 we show the performance of the equality function across different libraries, datatypes and compilation flags. From this graph, it is clear that having no optimization (level `O0`) is much worse than `O1`. From this point on we will omit the non-optimized figures. Inlining thresholds as high as `O1CT1000UT180` seem to have no added benefit, so we will also omit them.

Overall, `syb` performs poorly when compared to the other approaches. Standard optimization level `O1` gives a performance 10–15 times worse than hand-written depending on the datatype. Aggressive inlining reduces this figure to 7–12 times worse, which is still an enormous overhead. This is probably because of `syb`'s complex implementation for functions which traverse two generic arguments.

On the other hand, `emgm` performs almost as fast as hand-written for eq_{Tree} even when compiled with `O1` only. For eq_{Logic}, standard compilation gives a performance roughly 2 times slower than hand-written, but forcing inlining brings this down to 1 again. This is a remarkable improvement, and it shows that the benefits of stimulating inlining are not limited to `regular`, as both the results

[5] `http://www.cs.uu.nl/wiki/GenericProgramming/EMGM`

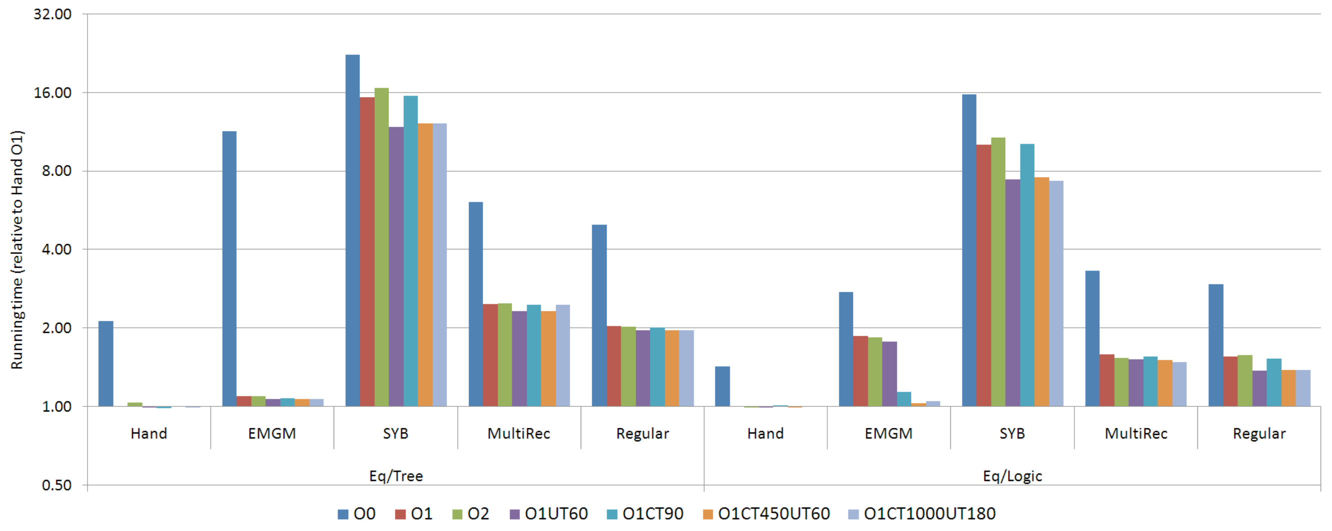

Figure 1. Performance of generic equality.

of syb and emgm show. For this test, multirec and regular have the least benefits from inlining, taking between 1.5 and 2.5 times longer than hand-written code.

It is interesting that the results for eq_{Logic} seem, in general, better than those for eq_{Tree}. This is caused by the test structure: while for eq_{Tree} we are comparing trees that either are equal or only differ at a deep leaf, for eq_{Logic} we also have some comparisons which return *False* rather quickly. This means less time is spent in the generic function, and therefore there is less overhead from generics.

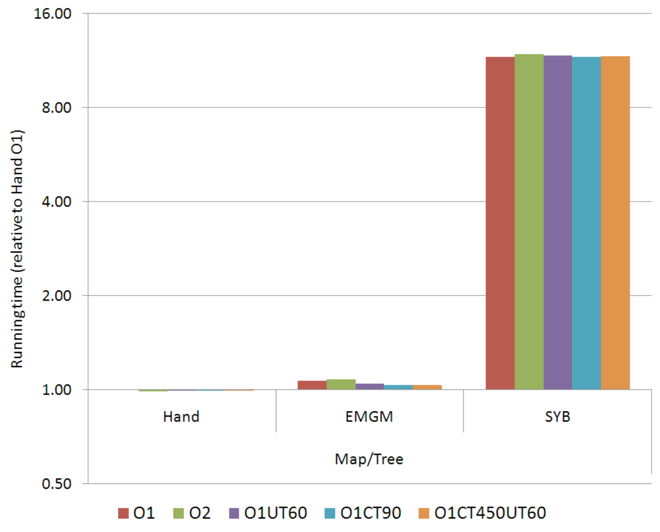

Figure 2. Performance of generic map.

Map In Figure 2 we show the performance of generic map. We only show the emgm and syb libraries, since regular and multirec cannot adequately express a map on the arguments of type containers. The syb test is not implemented as a truly generic map; instead we use the *everywhere* combinator for traversals. We only show the *Tree* datatype, since *Logic* is not a type container.

It is clear from this test that emgm's implementation of generic map is very well optimized, generating code that has an overhead of only 7% with standard O1. On the other hand, the implementation using syb's type-safe cast takes approximately 12 times longer to run, and shows no improvement with increased inlining thresholds.

Show We explored the potential for optimization of generic show in Section 3.2. Here we confirm those results by benchmarking. The results are in Figure 3. For this test, emgm and regular seem to have the most dramatic improvements. With O1CT450UT60, emgm has a penalty of only 10% over the standard hand-written version, while regular performs even slightly faster. In Section 3.2 we have seen that the optimizations lead to generation of code that is comparable to that generated for a hand-written version. The benchmark confirms this fact. Both syb and multirec do not seem to benefit as strongly from the inlining, although O1CT450UT60 still provides the best results. This also seems to hint at the fact that the added complexity of multirec (when compared to regular) complicates optimization. It is also interesting to note that the *Logic* datatype does not prove harder to handle for the generic libraries, even when not using a balanced sum-of-products view (Holdermans et al. 2006).

Read In Figure 4 we show the results of benchmarking generic read. Unlike generic show, reading does not seem to benefit much from inlining. Additionally, all libraries perform significantly worse than the hand-written version, with emgm, multirec and regular all performing roughly 8 times slower and syb 20, for the *Tree* datatype.

The tests for the *Logic* datatype give similar performance results for emgm and syb. However, both regular and multirec were unable to handle this test, consuming too much memory and grinding the machine to a halt. We assume this to be due to some inefficiency while parsing the alternatives in the sum case. This could be solved by adapting generic read on these libraries to behave more like emgm's.

Update Our last test is generic update (Figure 5). We notice several important figures:

- Optimization levels O1UT60 and O1CT450UT60 again impact performance considerably (for syb on the *Logic* datatype). Conversely, they produce the best results for regular (on both datatypes), which are as good as hand-written code.

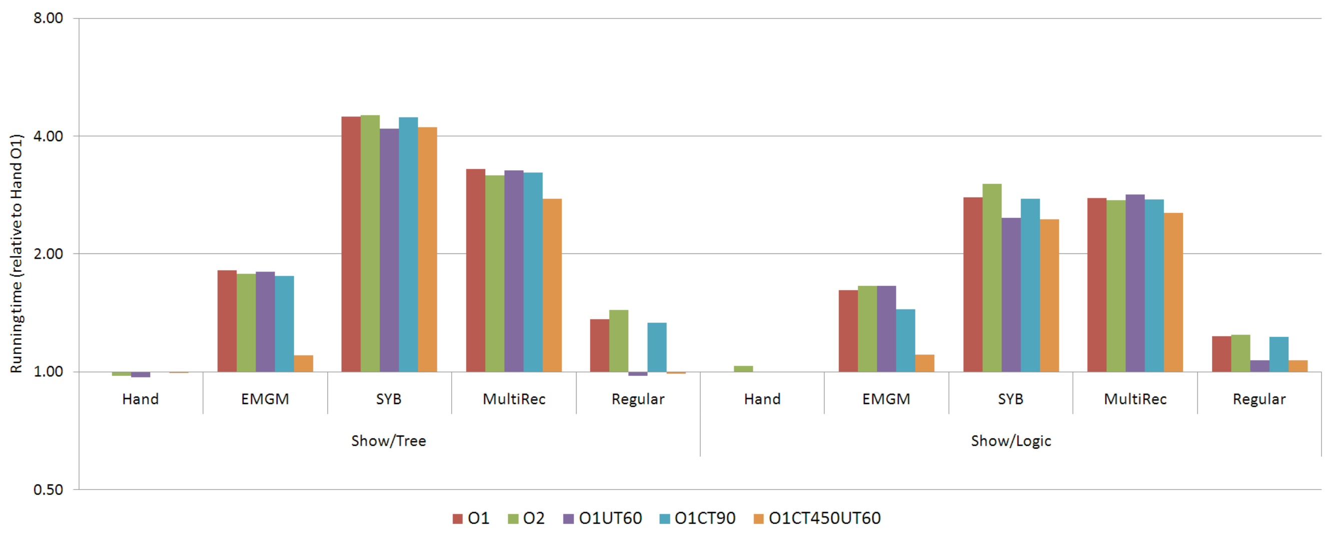

Figure 3. Performance of generic show.

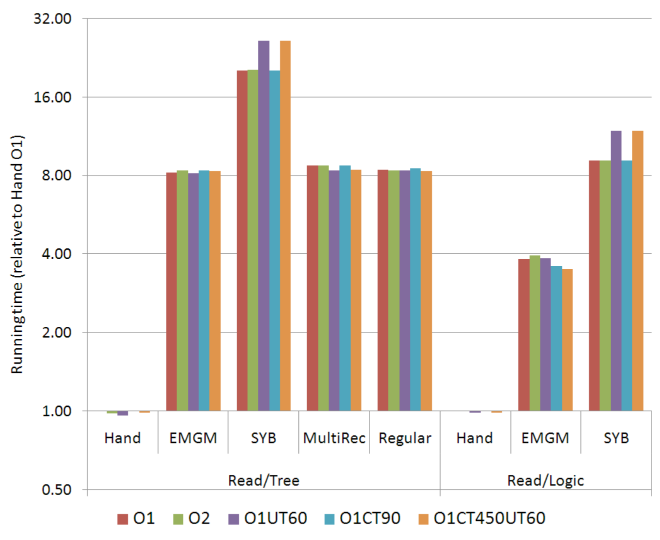

Figure 4. Performance of generic read.

- While `emgm` performs clearly worse than hand-written code for the *Logic* datatype when compiled with `O1` (5 times worse), the overhead seems to be completely removed by compiling with `O1CT450UT60`. The same does not hold for the *Tree* datatype, where both `O1` and `O1CT450UT60` give a similar penalty of 31%.
- From a user's perspective, while the implementation of the generic traversals in `emgm` and `syb` is very similar (in terms of the generic function *everywhere*), `syb` has much worse performance. This is probably due to the rank-2 polymorphism in the `syb` version:

$$\begin{aligned} everywhere_{\texttt{emgm}} &:: (Rep\ (Everywhere\ a)\ b) \\ &\Rightarrow (a \to a) \to b \to b \\ everywhere_{\texttt{syb}} &:: (Data\ a) \\ &\Rightarrow (\forall a.\ Data\ a \Rightarrow a \to a) \to a \to a\ . \end{aligned}$$

The `syb` version is more flexible, but this comes at the extra price of loss of optimization opportunities.

- Again, `multirec` performs worse than `regular`, and does not seem to be affected by the higher thresholds for inlining. In the test for *Logic*, `multirec` even performs slower than `syb`, making this the only test where `syb` is not the slowest library.

5.4.2 Operating systems

Since our benchmarking framework is independent of the operating system, we ran it on three different machines to check for possible differences in the optimization behavior. We summarize the results in Figure 6. This graph illustrates the average performance of all tests, each still relative to the hand-written variant compiled with `O1`. We do not show the labels of the y-axis since it has no significance, but it is linear (and not logarithmic as on the previous graphs). While we expected no major differences between the operating systems, it turned out that GHC optimizes less under Windows XP. The difference between Linux and MacOS is mostly insignificant, apart from the `O0` level.

In this graph we can also evaluate the overall performance gains of each optimization level. We see that there is hardly any difference between `O1` and `O2`, and all of the optimizations with increased inlining improve performance on average. The best results are achieved with `O1CT450UT60`, while further increasing the thresholds (`O1CT1000UT180`) decreases performance (even if only slightly). In any case, it should be noted that not many conclusions can be taken from the averages, since we have seen that blindly increasing inlining can negatively impact performance. It is insightful to notice that there is an overall gain, but this does not mean that the thresholds for inlining should always be raised.

5.5 Analysis

In this section we discuss the relevance of our results and compare them to other benchmarks.

From the results of our benchmark, it is important to extract some general advice to the writer and user of generic programming libraries concerned with performance. It is safe to say that, as far as performance is concerned, `syb` and `multirec` are best avoided. The advantages of `multirec`, such as a zipper operating on mutually recursive datatypes, are not found in any other library, but `syb` is many times used as a default, since it is readily available and comes with GHC-specific support. However, given the current evolution of Haskell libraries into the universal Hackage package

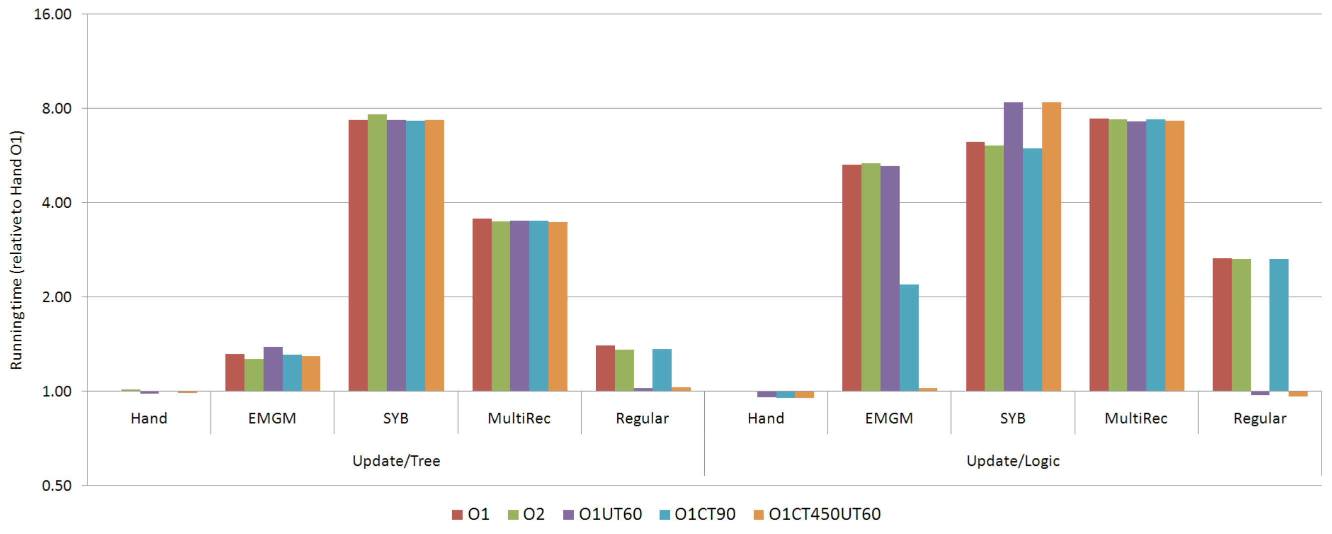

Figure 5. Performance of generic update.

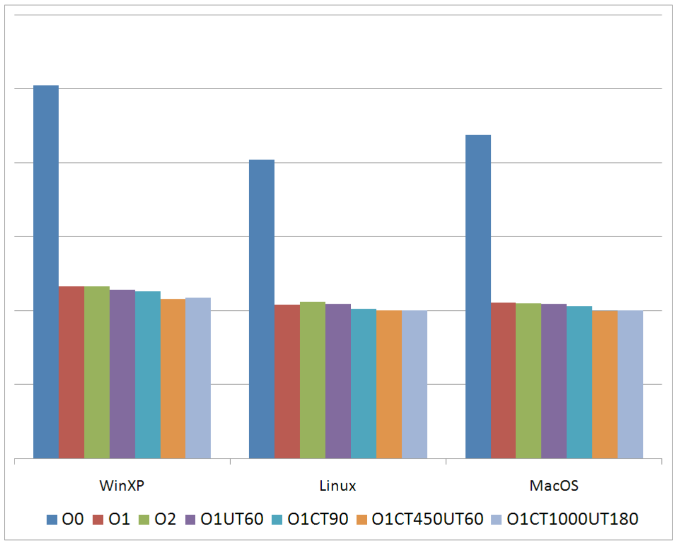

Figure 6. Performance across different operating systems.

repository[6], and the existence of maintained, well-documented and easy to use libraries such as `emgm` and `regular`, there are less reasons to default to `syb`.

We were somewhat surprised by the results of `multirec` in the benchmark. Being a library very similar to `regular` in terms of implementation, we expected similar performance results. However, not only `multirec` performs worse than `regular`, it also seems to benefit much less from increased inlining thresholds. This seems to indicate that explicit type witnesses and type equality constraints, which are frequent in `multirec`, prevent the elimination of the overhead from the generic code.

Unlike the other libraries which we benchmark, `syb` does not use a sum-of-products view on data, but instead the spine view (Hinze et al. 2006). Additionally, generic functions are not defined by induction on the representation types, but by use of primitive generic combinators, such as *gfoldl* and *gunfold*. These functions use rank-2 polymorphism and rely on run-time type-safe cast. These features set `syb` apart from the other libraries, and are most likely the cause for `syb`'s poor performance. Further research is necessary to improve the performance of `syb`, but we suspect that techniques other than inlining will be necessary for that.

As for users of `emgm` and `regular`, it is important to compile the code with the right flags. The library code should be compiled with high `CT`, so as to permit inlining of the generic functions at the use site. User code should be compiled with high `UT`, to force inlining to occur and, consequently, elimination of generic overhead through GHC's standard optimizations. We cannot uniquely determine the a single value to set these parameters to, since this will depend on the size of the functions. For now, the only way to determine these values is by inspection of the generated code or benchmarking.

We know of two other benchmarks of generic programs. Rodriguez Yakushev (2009, Chapter 4) benchmarks `emgm`, `syb` and `multirec` (among others) against hand-written code. From the three functions benchmarked, one is also in our benchmark (generic equality). The relative ordering of the performance of each library matches our benchmark (with `syb` being the slowest and `emgm` the fastest), but the results are significantly different. However, we have seen (Figure 1) that the structure of the datatype being compared affects the performance, so this is not surprising.

Brown and Sampson (2009) developed a specialized library tailored for good performance, `alloy`. The techniques used for good performance are, however, very specific and do not readily translate to other approaches. In fact, `alloy` cannot even express the generic equality function, being focused mostly on single parameter traversals. In their benchmark, `alloy` has a mixed performance when compared to `emgm`, varying between a quarter slower to a third faster, depending on the traversal. An important observation of Brown and Sampson is that `emgm` and `syb` should always use a specialized case for traversal of *String*s. Treating them as mere lists of characters can dramatically impact performance.

We have seen that increased thresholds for inlining can negatively impact performance. Other known side effects are code bloat and increased compilation times. We measured the average increase of binary size when using `O1CT450UT60`, with `O1` as a reference. We found an average increase of 1.6%, with a standard deviation of 0.7%. Compilation time increased 12%. We consider this acceptable for the gains that are achieved.

[6] `http://hackage.haskell.org`

6. Conclusion

We have presented a benchmark of generic programs, highlighting the importance of inlining for performance of generics. Our benchmark not only reveals which libraries are more or less fast, but also which compilation flags can dramatically affect performance. Although our benchmark answered many questions, it probably raised even more new questions. Tweaking the inlining flags is certainly not the most convenient way of optimizing generics, and it requires extra work for both library writers and users. Additionally, currently its effects cannot be restricted to single functions by means other than isolating the function in a separate module and separate compilation. This is rather undesirable. Ideally, library writers flag the generic functions which are to be inlined, while users get the best optimization without having to worry about setting flags with long and complicated names. It remains to see why the `INLINE` pragma is not sufficient to achieve this behavior, and which alternatives could be implemented. An alternative is to use rewrite rules to encode inlining. Rules such as

$$
\begin{array}{l}
\texttt{\{-\# RULES "fromLogic"} \ \forall x.\\
\quad \mathit{from}\ x = \textbf{case}\ x\ \textbf{of}\\
\qquad\qquad \begin{array}{lll}
(p \vee q) & \rightarrow L & (C\ ((I\ p) \times (I\ q)))\\
(\mathit{Var}\ x) & \rightarrow R\ (L & (C\ (K\ x)))\\
(\mathit{Not}\ p) & \rightarrow R\ (R\ (L\ (C\ (I\ p)))) & \\
(\mathit{Const}\ b) & \rightarrow R\ (R\ (R\ (C\ (K\ b)))) & \texttt{\#-\}}
\end{array}
\end{array}
$$

could be used for that purpose. It remains to be seen if they could achieve similar performance to that achieved by tweaking inlining flags. Naturally, such rules would have to be automatically generated to remain unobtrusive.

While `regular` and `emgm` have shown promising results with increased inlining for most generic functions, it remains to see how to optimize generic read. Additionally, other generic producers might also prove problematic to optimize. For `multirec`, it remains to be seen which of its advanced techniques are preventing optimization through inlining, and how to circumvent that. For `syb`, the slowest and most popular generic programming library in Haskell, an entire new approach to optimization is necessary. We plan to investigate all of these in our future research.

In any case, it is now clear that generic programs do not have to be slow, and their optimization up to hand-written code performance not only is possible but can also be done with standard optimization techniques for functional programs. This opens the door for a future where generic programs will not only be general, elegant, and concise but also as efficient as type-specific code.

A. Configuration of the test machines

The benchmark figures were obtained from a Linux machine running on kernel version 2.6.28.10 on an Intel Core 2 Duo 2.13Ghz with 4GB of RAM. For the comparison across operating systems, we also ran the benchmark on a Windows machine running XP Professional x64 Edition with SP2 on an Intel Core 2 Duo 3Ghz with 2GB of RAM, and on a Macintosh running Mac OS X 10.5.7 on an Intel Core Duo 1.66Ghz with 1GB of RAM. All machines were running GHC version 6.10.4, the latest released version at the time of writing. All raw data used for this paper is available together with the benchmark suite at `https://subversion.cs.uu.nl/repos/staff.jpm.public/benchmark`.

Acknowledgments

This work has been partially funded by the Portuguese Foundation for Science and Technology (FCT) via the SFRH/BD/35999/2007 grant, and by the Netherlands Organisation for Scientific Research (NWO) through its projects on “Scriptable Compilers” (612.063.406) and “Real-life Datatype-Generic programming” (612.063.613). We thank Alexey Rodriguez and Erik Hesselink for the stimulating discussions on this subject.

References

Artem Alimarine and Sjaak Smetsers. Optimizing generic functions. In Dexter Kozen and Carron Shankland, editors, *MPC*, volume 3125 of *LNCS*, pages 16–31. Springer, 2004.

Roland Backhouse, Patrik Jansson, Johan Jeuring, and Lambert Meertens. Generic programming—an introduction. In *AFP'98*, volume 1608 of *LNCS*, pages 28–115. Springer, 1999.

Neil C. C. Brown and Adam T. Sampson. Alloy: Fast generic transformations for Haskell. In *Haskell'09*, 2009.

Jeremy Gibbons. Datatype-generic programming. In Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring, editors, *Spring School on Datatype-Generic Programming*, volume 4719 of *LNCS*. Springer, 2007.

R. Hinze, J. Jeuring, and A. Löh. Comparing approches to generic programming in Haskell. In *Datatype-Generic Programming*, LNCS 4719, pages 72–149. Springer, 2007.

Ralf Hinze. Generics for the masses. *Journal of Functional Programming*, 16(4-5):451–483, 2006.

Ralf Hinze, Andres Löh, and Bruno C. d. S. Oliveira. “Scrap Your Boilerplate” reloaded. In Philip Wadler and Masimi Hagiya, editors, *Proceedings of the 8th International Symposium on Functional and Logic Programming, FLOPS 2006*, volume 3945 of *LNCS*. Springer, 2006.

Stefan Holdermans, Johan Jeuring, Andres Löh, and Alexey Rodriguez Yakushev. Generic views on data types. In Tarmo Uustalu, editor, *MPC*, volume 4014 of *LNCS*, pages 209–234. Springer, 2006.

Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical approach to generic programming. In *TLDI '03*, pages 26–37, 2003.

Ralf Lämmel and Simon Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *ICFP 2004*, pages 244–255. ACM, 2004.

Thomas van Noort, Alexey Rodriguez Yakushev, Stefan Holdermans, Johan Jeuring, and Bastiaan Heeren. A lightweight approach to datatype-generic rewriting. In *WGP '08*, pages 13–24. ACM, 2008.

Bruno C.d.S. Oliveira, Ralf Hinze, and Andres Löh. Extensible and modular generics for the masses. In Henrik Nilsson, editor, *Trends in Functional Programming (TFP 2006)*, April 2007.

Will Partain. The `nofib` Benchmark Suite of Haskell Programs. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pages 195–202, London, UK, 1993. Springer.

Simon Peyton Jones and Simon Marlow. Secrets of the Glasgow Haskell Compiler inliner. *Journal of Functional Programming*, 12(4&5):393–433, 2002.

Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the Rules: Rewriting as a practical optimisation technique in GHC. In Ralf Hinze, editor, *Haskell'01*, page 203, 2001.

Simon Peyton Jones et al. *Haskell 98, Language and Libraries. The Revised Report*. Cambridge University Press, 2003. A special issue of JFP.

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

Alexey Rodriguez Yakushev, Johan Jeuring, Patrik Jansson, Alex Gerdes, Oleg Kiselyov, and Bruno C.d.S. Oliveira. Comparing libraries for generic programming in Haskell. In *Haskell'08*, pages 111–122, 2008.

Alexey Rodriguez Yakushev, Stefan Holdermans, Andres Löh, and Johan Jeuring. Generic programming with fixed points for mutually recursive datatypes. In *ICFP 2009*, 2009.

Tim Sheard and Simon Peyton Jones. Template metaprogramming for Haskell. In Manuel M. T. Chakravarty, editor, *Haskell'02*, pages 1–16. ACM, 2002.