# Optimisation of Generic Programs through Inlining

José Pedro Magalhães*

Department of Computer Science, University of Oxford
jpm@cs.ox.ac.uk

**Abstract.** It is known that datatype-generic programs often run slower than type-specific variants, and this factor can prevent adoption of generic programming altogether. There can be multiple reasons for the performance penalty, but often it is caused by conversions to and from representation types that do not get eliminated during compilation. However, it is also known that generic functions can be specialised to specific datatypes, removing any overhead from the use of generic programming. In this paper, we investigate compilation techniques to specialise generic functions and remove the performance overhead of generic programs in Haskell. We pick a representative generic programming library and look at the generated code for a number of example generic functions. After understanding the necessary compiler optimisations for producing efficient generic code, we benchmark the runtime of our generic functions against handwritten variants, and conclude that the overhead can indeed be removed automatically by the compiler.

## 1 Introduction

Datatype-generic programming is a form of abstraction that allows defining functions that operate on every suitable datatype. Generic programs operate on the general structure of datatypes, therefore remaining agnostic of the individual detail of each datatype. Examples of behaviour that can be defined generically are (de)serialisation, equality testing, and traversing data. It is convenient to define such functions generically because less code has to be written, and this code has to be adapted less often. However, generic programs operate on the underlying structure of datatypes, and not on datatypes themselves directly. This indirection often causes a runtime penalty, as conversions to and from the generic representation are not always optimised away.

The performance of generic programs has been analysed before. Rodriguez Yakushev et al. (2008) present a detailed comparison of nine libraries for generic programming in Haskell, with 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. (2010) 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 structure editors or compilers.

* This work has been funded by EPSRC grant number EP/J010995/1. We thank the anonymous reviewers for the helpful feedback.

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 datatypes by a small set of representation types. Equipped with conversion functions between user datatypes and their representation, we can define functions on the representation types, which are then applicable to all user types via the conversion functions. While these conversion functions are typically trivial and can be automatically generated, the overhead they impose is not automatically removed. In general, conversions to and from the generic representations are not eliminated by compilation, and are performed at run-time. These conversions are the main source of inefficiency for generic programming libraries. In the earlier implementations of generic programming as code generators or preprocessors (Hinze et al. 2007), optimisations (such as automatic generation of type-specialised variants of generic functions) could be implemented externally. Modern implementations of generic programming are libraries, removing the need for cumbersome work on parsing and type checking, for instance. With the switch to library approaches, however, all optimisations have to be performed by the compiler.

The Glasgow Haskell Compiler (GHC, the main Haskell compiler) compiles a program by first converting the input into a core language and then transforming the core code into more optimised versions, in a series of sequential passes. While it performs a wide range of optimisations, with the default settings it seems to be unable to remove the overhead incurred by using generic representations. Therefore generic libraries perform worse than handwritten type-specific counterparts. Alimarine and Smetsers (2004, 2005) show that in many cases it is possible to remove all overhead by performing a specific form of symbolic evaluation in the Clean compiler. In fact, their approach is not restricted to optimising generics, and GHC performs symbolic evaluation as part of its optimisations. Our goal is to convince GHC to optimise generic functions so as to achieve the same performance as handwritten code, without requiring any additional manipulation of the compiler internals.

We have investigated this problem before (Magalhães et al. 2010), and concluded that tweaking GHC optimisation flags can achieve significant speedups. The problem with using compiler flags is that these apply to the entire program being compiled, and while certain flags might have a good effect on generic functions, they might adversely affect performance (or code size) of other parts of the program. In this paper we take a more fine-grained approach to the problem, looking at how to localise our performance annotations to the generic code only, by means of rewrite rules and function pragmas.[1] In this way we can improve the performance of generic functions with minimal impact on the rest of the program.

We continue this paper by defining two representative generic functions which we focus our optimisation efforts on (Section 2). We then see how these functions can be optimised manually (Section 3), and transfer the necessary optimisation techniques to the compiler (Section 4). We confirm that our optimisations result in better runtime performance of generic programs in a benchmark in Section 5, and conclude in Section 6.

[1] `http://www.haskell.org/ghc/docs/7.4.1/html/users_guide/pragmas.html`

## 2 Example generic functions

For analysing the performance of generic programs we choose the `generic-deriving` library, now integrated in GHC. Due to space considerations we can only provide the (simplified) interface of this library:

$$
\begin{array}{ll}
\textbf{data}\ U_1 \quad \rho = U_1 & \textbf{class}\ \mathit{Generic}\ \alpha\ \textbf{where} \\
\textbf{data}\ K_1\ \alpha \quad \rho = K_1\ \alpha & \quad \textbf{type}\ \mathit{Rep}\ \alpha :: \star \to \star \\
\textbf{data}\ (\alpha :+: \beta)\ \rho = L_1\ (\alpha\ \rho) \mid R_1\ (\beta\ \rho) & \quad \mathit{to} \quad :: \mathit{Rep}\ \alpha\ \rho \to \alpha \\
\textbf{data}\ (\alpha :\times: \beta)\ \rho = \alpha\ \rho :\times: \beta\ \rho & \quad \mathit{from} :: \alpha \to \mathit{Rep}\ \alpha\ \rho
\end{array}
$$

$U_1$ encodes constructors without arguments. $K_1\ \alpha\ \rho$ encodes recursion into some datatype $\alpha$. Finally, $(:+:)$ encodes choice between constructors, and $(:\times:)$ is used for constructors with multiple arguments. The parameter $\rho$, present in all the representation types, is not used by our example generic functions and can be safely ignored. The type class *Generic* encodes the conversion between a datatype $\alpha$ and its representation $\mathit{Rep}\ \alpha$, witnessed by the conversion functions *to* and *from*. The reader is referred to Magalhães (2012) for a full description of `generic-deriving`.

We present two generic functions that will be the focus of our attention: equality and enumeration. These are chosen as representative examples; equality is a generic consumer, taking generic values as input, and enumeration is a generic producer, since it generates generic values. Equality is a relatively simple, standard example, while enumeration requires the use of auxiliary (non-generic) functions.

### 2.1 Generic equality

A notion of structural equality can easily be defined as a generic function. We first define a class for equality on the representation types:

$$
\begin{array}{l}
\textbf{class}\ \mathit{GEqRep}\ \phi\ \textbf{where} \\
\quad \mathit{geqRep} :: \phi\ \alpha \to \phi\ \alpha \to \mathit{Bool}
\end{array}
$$

We can now give instances for each of the representation types:

$$
\begin{array}{l}
\textbf{instance}\ \mathit{GEqRep}\ U_1\ \textbf{where} \\
\quad \mathit{geqRep}\ _\ _ = \mathit{True} \\
\textbf{instance}\ (\mathit{GEqRep}\ \alpha, \mathit{GEqRep}\ \beta) \Rightarrow \mathit{GEqRep}\ (\alpha :+: \beta)\ \textbf{where} \\
\quad \mathit{geqRep}\ (L_1\ x)\ (L_1\ y) = \mathit{geqRep}\ x\ y \\
\quad \mathit{geqRep}\ (R_1\ x)\ (R_1\ y) = \mathit{geqRep}\ x\ y \\
\quad \mathit{geqRep}\ _\ \ _ = \mathit{False} \\
\textbf{instance}\ (\mathit{GEqRep}\ \alpha, \mathit{GEqRep}\ \beta) \Rightarrow \mathit{GEqRep}\ (\alpha :\times: \beta)\ \textbf{where} \\
\quad \mathit{geqRep}\ (x_1 :\times: y_1)\ (x_2 :\times: y_2) = \mathit{geqRep}\ x_1\ x_2 \wedge \mathit{geqRep}\ y_1\ y_2
\end{array}
$$

Units are trivially equal. For sums we continue the comparison recursively if both values are either on the left or on the right, and return *False* otherwise. Products are equal if both components are equal.

For recursive occurrences we fall back to a user-facing *GEq* class:

$$
\begin{array}{l}
\textbf{instance}\ (\mathit{GEq}\ \gamma) \Rightarrow \mathit{GEqRep}\ (K_1\ \gamma)\ \textbf{where} \\
\quad \mathit{geqRep}\ (K_1\ a)\ (K_1\ b) = \mathit{geq}\ a\ b
\end{array}
$$

This user-facing class is similar to *GEqRep*, but is used for user datatypes, and comes with a generic default method:

```
class GEq α where
  geq :: α → α → Bool
  default geq :: (Generic α, GEqRep (Rep α)) ⇒ α → α → Bool
  geq x y = geqRep (from x) (from y)
```

This class is similar to the Prelude *Eq* class, but we have left out inequality for simplicity. The generic default simply calls *from* on the arguments, and then proceeds using the generic equality function *geqRep*.

Adhoc instances for base types can reuse the Prelude implementation:

```
instance GEq Int where
  geq = (≡)
```

User datatypes, such as lists, can use the generic default:

```
instance (GEq α) ⇒ GEq [α]
```

### 2.2 Generic enumeration

We now define a function that enumerates all possible values of a datatype. For infinite datatypes we have to make sure that every possible value will eventually be produced. For instance, if we are enumerating integers, we should not first enumerate all positive numbers, and then the negatives. Instead, we should interleave positive and negative numbers.

We enumerate values by listing them with the standard list type. There is only one unit to enumerate, and for datatype occurrences we refer to a user-facing *GEnum* class:

```
class GEnumRep ϕ where
  genumRep :: [ϕ α]

instance GEnumRep U1 where
  genumRep = [U1]
instance (GEnum α) ⇒ GEnumRep (K1 α) where
  genumRep = map K1 genum
```

The more interesting cases are those for sums and products. For sums we enumerate both alternatives, but interleave them with a (|||) operator:

```
instance (GEnumRep α, GEnumRep β) ⇒ GEnumRep (α :+: β) where
  genumRep = map L1 genumRep ||| map R1 genumRep

infixr 5 |||
(|||) :: [α] → [α] → [α]
```

For products we generate all possible combinations of the two arguments, and diagonalise the result matrix, ensuring that all elements from each sublist will eventually be included, even if the lists are infinite:

$$\begin{array}{l}\textbf{instance}\ (\mathit{GEnumRep}\ \alpha, \mathit{GEnumRep}\ \beta) \Rightarrow \mathit{GEnumRep}\ (\alpha :\times: \beta)\ \textbf{where} \\ \quad \mathit{genumRep} = \mathit{diag}\ (\mathit{map}\ (\lambda x \to \mathit{map}\ (\lambda y \to x :\times: y)\ \mathit{genumRep})\ \mathit{genumRep})\end{array}$$

$$\mathit{diag} :: [\,[\,\alpha\,]\,] \to [\,\alpha\,]$$

We omit the implementation details of $(|||)$ and *diag* as they are not important; it only matters that we have some form of fair interleaving and diagonalisation operations. The presence of $(|||)$ and *diag* throughout the generic function definition makes enumeration more complicated than equality, since equality does not make use of any auxiliary functions. We will see in Section 4.3 how this complicates the specialisation process. Note also that we do not use the more natural list comprehension syntax for defining the product instance, again to simplify the analysis of the optimisation process.

Finally, we define the user-facing class, with a default implementation:

$$\begin{array}{l}\textbf{class}\ \mathit{GEnum}\ \alpha\ \textbf{where} \\ \quad \mathit{genum} :: [\,\alpha\,] \\ \quad \textbf{default}\ \mathit{genum} :: (\mathit{Generic}\ \alpha, \mathit{GEnumRep}\ (\mathit{Rep}\ \alpha)) \Rightarrow [\,\alpha\,] \\ \quad \mathit{genum} = \mathit{map\ to\ genumRep}\end{array}$$

## 3 Specialisation, by hand

We now focus on the problem of specialisation of generic functions. By specialisation we mean removing the use of generic conversion functions and representation types, replacing them by constructors of the original datatype. To convince ourselves that this task is possible, we first develop a hand-written derivation of specialisation by equational reasoning. For simplicity we ignore implementation mechanisms such as the use of type classes and type families, and focus first on a very simple datatype encoding natural numbers:

$$\textbf{data}\ \mathit{Nat} = \mathit{Ze} \mid \mathit{Su}\ \mathit{Nat}$$

We give the representation of naturals with standard Haskell datatypes using a type synonym:

$$\textbf{type}\ \mathit{RepNat} = \mathit{Either}\ ()\ \mathit{Nat}$$

We use a shallow representation (with *Nat* at the leaves, and not *RepNat*), remaining faithful with `generic-deriving`. We also need a way to convert between *RepNat* and *Nat*:

$$\begin{array}{l}\mathit{toNat} :: \mathit{RepNat} \to \mathit{Nat} \\ \mathit{toNat}\quad n = \textbf{case}\ n\ \textbf{of}\ \{\mathit{Left}\ () \to \mathit{Ze};\ \mathit{Right}\ n \to \mathit{Su}\ n;\}\end{array}$$

$$\begin{array}{l}\mathit{fromNat} :: \mathit{Nat} \to \mathit{RepNat} \\ \mathit{fromNat}\ n = \textbf{case}\ n\ \textbf{of}\ \{\mathit{Ze} \to \mathit{Left}\ ();\ \mathit{Su}\ n \to \mathit{Right}\ n;\}\end{array}$$

We now analyse the specialisation of generic equality and enumeration on this datatype.

### 3.1 Generic equality

We consider two versions of an equality function. The first is a handwritten, type-specific definition of equality for *Nat*:

```
eqNat :: Nat → Nat → Bool
eqNat m n = case (m , n) of
                (Ze    , Ze   ) → True
                (Su m  , Su n) → eqNat m n
                (_     , _    ) → False
```

The second is generic equality on *Nat* through *RepNat*, for which we need equality on units and sums:

```
eqU :: () → () → Bool
eqU x y = case (x , y) of { (() , ()) → True; }

eqPlus :: (α → α → Bool) → (β → β → Bool) → Either α β → Either α β → Bool
eqPlus ea eb a b = case (a , b) of
                       (Left  x , Left  y) → ea x y
                       (Right x , Right y) → eb x y
                       (_       , _       ) → False
```

Now we can define equality for *RepNat*, and generic equality for *Nat* through conversion to *RepNat*:

```
eqRepNat :: RepNat → RepNat → Bool
eqRepNat = eqPlus eqU eqNatFromRep

eqNatFromRep :: Nat → Nat → Bool
eqNatFromRep m n = eqRepNat (fromNat m) (fromNat n)
```

Our goal now is to show that *eqNatFromRep* is equivalent to *eqNat*. In the following derivation, we start with the definition of *eqNatFromRep*, and end with the definition of *eqNat*:

```
  eqRepNat (fromNat m) (fromNat n)

≡⟨ inline eqRepNat and eqPlus ⟩

  case (fromNat m , fromNat n) of
    (Left  x , Left  y) → eqU x y
    (Right x , Right y) → eqNatFromRep x y
    _                   → False

≡⟨ inline fromNat ⟩

  case (  case m of {Ze → Left (); Su x1 → Right x1}
        , case n of {Ze → Left (); Su x2 → Right x2}) of
    (Left  x , Left  y) → eqU x y
    (Right x , Right y) → eqNatFromRep x y
    _                   → False
```

$\equiv\langle$ case-of-case transform $\rangle$

$$
\begin{array}{l}
\textbf{case}\ (m\ ,\ n)\ \textbf{of} \\
\quad (Ze\ \ ,\ Ze\ \ ) \rightarrow eqU\ ()\ () \\
\quad (Su\ x_1\ ,\ Su\ x_2) \rightarrow eqNatFromRep\ x_1\ x_2 \\
\quad _ \qquad\qquad\qquad \rightarrow False
\end{array}
$$

$\equiv\langle$ inline $eqU$ and case-of-constant $\rangle$

$$
\begin{array}{l}
\textbf{case}\ (m\ ,\ n)\ \textbf{of} \\
\quad (Ze\ \ ,\ Ze\ \ ) \rightarrow True \\
\quad (Su\ x_1\ ,\ Su\ x_2) \rightarrow eqNatFromRep\ x_1\ x_2 \\
\quad _ \qquad\qquad\qquad \rightarrow False
\end{array}
$$

This shows that the generic implementation is equivalent to the type-specific variant, and that it can be optimised to remove all conversions. We discuss the techniques used in this derivation in more detail in Section 4.1, after showing the optimisation of generic enumeration.

### 3.2 Generic enumeration

A type-specific enumeration function for *Nat* follows:

$$
\begin{array}{l}
enumNat :: [Nat] \\
enumNat = [Ze] \mid\mid\mid map\ Su\ enumNat
\end{array}
$$

To get an enumeration for *RepNat* we first need to know how to enumerate units and sums:

$$
\begin{array}{l}
enumU :: [()] \\
enumU = [()] \\
enumPlus :: [\alpha] \rightarrow [\beta] \rightarrow [Either\ \alpha\ \beta] \\
enumPlus\ ea\ eb = map\ Left\ ea \mid\mid\mid map\ Right\ eb
\end{array}
$$

Now we can define an enumeration for *RepNat*:

$$
\begin{array}{l}
enumRepNat :: [RepNat] \\
enumRepNat = enumPlus\ enumU\ enumNatFromRep
\end{array}
$$

With the conversion function *toNat*, we can use *enumRepNat* to get a generic enumeration function for *Nat*:

$$
\begin{array}{l}
enumNatFromRep :: [Nat] \\
enumNatFromRep = map\ toNat\ enumRepNat
\end{array}
$$

We now show that *enumNatFromRep* and *enumNat* are equivalent:[2]

[2] Given that these are recursive structures, we have to be careful to preserve correctness over the whole proof, even if each step is clearly correct (Sands 1998). None of the steps in the proof changes the productivity of the entire expression, so we are confident of its overall correctness.

$$
\begin{aligned}
& map\ toNat\ enumRepNat \\
\equiv\langle\ & \text{inline } enumRepNat \text{ and } enumPlus\ \rangle \\
& map\ toNat\ (map\ Left\ enumU \mid\mid\mid map\ Right\ enumNatFromRep) \\
\equiv\langle\ & \text{inline } enumU \text{ and } map\ \rangle \\
& map\ toNat\ ([Left\ ()] \mid\mid\mid map\ Right\ enumNatFromRep) \\
\equiv\langle\ & \text{free theorem } (\mid\mid\mid) : \forall f\ a\ b.map\ f\ (a \mid\mid\mid b) = map\ f\ a \mid\mid\mid map\ f\ b\ \rangle \\
& map\ toNat\ [Left\ ()] \mid\mid\mid map\ toNat\ (map\ Right\ enumNatFromRep) \\
\equiv\langle\ & \text{inline } map \text{ and } toNat, \text{ case-of-constant}\ \rangle \\
& [Ze] \mid\mid\mid map\ toNat\ (map\ Right\ enumNatFromRep) \\
\equiv\langle\ & \text{functor composition law: } \forall f\ g\ l.map\ f\ (map\ g\ l) = map\ (f \circ g)\ l\ \rangle \\
& [Ze] \mid\mid\mid map\ (toNat \circ Right)\ enumNatFromRep \\
\equiv\langle\ & \text{inline } toNat \text{ and case-of-constant}\ \rangle \\
& [Ze] \mid\mid\mid map\ Su\ enumNatFromRep
\end{aligned}
$$

Like equality, generic enumeration can also be specialised to a type-specific variant without any overhead.

## 4 Specialisation, by the compiler

After the manual specialisation of generic functions, let us now analyse how to convince the compiler to automatically perform the specialisation.

### 4.1 Optimisation techniques

Our calculations in Section 3 rely on a number of lemmas and techniques that the compiler will have to use. We review them here:

*Inlining* Inlining replaces a function call with its definition. It is a crucial optimisation technique because it can expose other optimisations. However, inlining causes code duplication, and care has to be taken to avoid non-termination through infinite inlining.

GHC uses a number of heuristics to decide when to inline a function or not, and loop breakers for preventing infinite inlining (Peyton Jones and Marlow 2002). The programmer can provide explicit inlining annotations with the *INLINE* and *NOINLINE* pragmas, of the form:

$$\{-\#\ INLINE\ [n]\ f\ \#-\}$$

In this pragma, $f$ is the function to be inlined, and $n$ is a phase number. GHC performs a number of optimisation phases on a program, numbered in decreasing order until zero.

Setting $n$ to 1, for instance, means "be keen to inline $f$ in phase 1 and after". For a *NOINLINE* pragma, this means "do not inline $f$ in phase 1 or after". The phase can be left out, in which case the pragma applies to all phases.[3]

*Application of free theorems and functor laws* Free theorems (Wadler 1989) are theorems that arise from the type of a polymorphic function, regardless of the function's definition. Each polymorphic function is associated with a free theorem, and functions with the same type share the same theorem. The functor laws arise from the categorical nature of functors. Every *Functor* instance in Haskell should obey the functor laws.

GHC does not compute and use the free theorem of each polymorphic function, in particular because it may not be clear which direction of the theorem is useful for optimisation purposes. However, we can add special optimisation rules to GHC via a *RULES* pragma (Peyton Jones et al. 2001). For instance, the rewrite rule corresponding to the free theorem of $(|||)$ follows:

$$\{-\#\ \mathit{RULES}\ \texttt{"ft/|||"}\ \forall f\ a\ b.\ \mathit{map}\ f\ (a\,|||\,b) = \mathit{map}\ f\ a\,|||\,\mathit{map}\ f\ b\ \#-\}$$

This pragma introduces a rule named "ft/|||" telling GHC to replace occurrences of the application $\mathit{map}\ f\ (a\,|||\,b)$ with $\mathit{map}\ f\ a\,|||\,\mathit{map}\ f\ b$. GHC does not perform any confluence checking on rewrite rules, so the programmer should ensure confluence or GHC might loop during compilation.

*Optimisation of case statements* Case statements drive evaluation in GHC's core language, and give rise to many possible optimisations. Peyton Jones and Santos (1998) provide a detailed account of these; in our derivation in Section 3.2 we used a "case of constant" rule to optimise a statement of the form:

$$\textbf{case}\ (\mathit{Left}\ ())\ \textbf{of}\ \{\mathit{Left}\ () \to \mathit{Ze};\ \mathit{Right}\ n \to \mathit{Su}\ n;\}$$

Since we know what we are case-analysing, we can replace this case statement by the much simpler expression $\mathit{Ze}$. Similarly, in Section 3.1 we used a case-of-case transform to eliminate an inner case statement. Consider an expression of the form:

$$\textbf{case}\ (\textbf{case}\ x\ \textbf{of}\ \{p_1 \to e_2;\})\ \textbf{of}\ \{p_2 \to e_3;\}$$

Here, $p_1$ and $p_2$ are patterns, $e_2$ and $e_3$ are expressions, and $e_2$ matches $p_2$. Taking care to avoid variable capture, we can often simplify this to:

$$\textbf{case}\ x\ \textbf{of}\ \{p_1 \to e_3;\}$$

This rule naturally generalises to case statements with multiple branches.

### 4.2 Generic equality

We have seen that we have a good number of tools at our disposal for directing the optimisation process in GHC: inline pragmas, rewrite rules, phase distinction, and all the standard optimisations for the functional core language. We will now annotate our generic functions and evaluate the quality of the core code generated by GHC.

[3] See the GHC User's Guide for more details: `http://www.haskell.org/ghc/docs/7.4.1/html/users_guide/pragmas.html`.

We start by defining a *Generic* instance for the *Nat* type:

```
instance Generic Nat where
  type Rep Nat = U1 :+: K1 Nat
  {-# INLINE [1] to #-}
  to (L1 U1)      = Ze
  to (R1 (K1 n)) = Su n
  {-# INLINE [1] from #-}
  from Ze      = L1 U1
  from (Su n) = R1 (K1 n)
```

We give inline pragmas for *to* and *from* to guarantee that these functions will be inlined. However, we ask the inliner to only inline them on phase 1 and after; this is to ensure that we first inline the generic function definitions, simplify those, and then inline the conversion functions and simplify again.

We can now provide a generic definition of equality for *Nat*:

```
instance GEq Nat
```

Compiling this code with the standard optimisation flag `-O` gives us the following core code:

```
$GEqNat_geq :: Nat → Nat → Bool
$GEqNat_geq = λ(x :: Nat) (y :: Nat) →
              case x of
                Ze    → case y of {Ze → True;  Su m → False; }
                Su m → case y of {Ze → False; Su n  → $GEqNat_geq m n; }
```

The core language is a small, explicitly typed language in the style of System F (Yorgey et al. 2012). The function $\$GEqNat_{geq}$ is prefixed with a \$ because it was generated by the compiler, representing the *geq* method of the *GEq* instance for *Nat*. We can see that the generic representation was completely removed.

The same happens for lists, as evidenced by the generated core code:

```
$GEq[]_geq :: ∀α.GEq α ⇒ [α] → [α] → Bool
$GEq[]_geq = λα (eqA :: GEq α) (l1 :: [α]) (l2 :: [α]) →
             case l1 of
               []        → case l2 of {[] → True; (h : t) → False; }
               (h1 : t1) → case l2 of
                             []        → False
                             (h2 : t2) → case eqA h1 h2 of
                                           False → False
                                           True  → $GEq[]_geq α eqA t1 t2
```

Note that type abstraction and application is explicit in core. There is syntax to distinguish type and value application and abstraction from each other, but we suppress the distinction since it is clear from the use of Greek letters for type variables. Note also that

constraints (to the left of the $\Rightarrow$ arrow) become just ordinary parameters, so $\$GEq[]_{geq}$ takes a function to compute equality on the list elements, *eqA*.[4]

Perhaps surprisingly, GHC performs all the required steps of Section 3.1 without requiring any annotations to the generic function itself. In general, however, we found that it is sensible to provide *INLINE* pragmas for each instance of the representation datatypes when defining a generic function. In the case of *geqRep*, the methods are small, so GHC inlines them eagerly. For more complicated generic functions, the methods may become larger, and GHC will avoid inlining them. Supplying an *INLINE* pragma tells GHC to inline the methods anyway.

### 4.3 Generic enumeration

Generic consumers, such as equality, are, in our experience, more easily optimised by GHC. A generic producer such as enumeration, in particular, is challenging because it requires map fusion, and lifting auxiliary functions through maps using free theorems. As such, we encounter some difficulties while optimising enumeration. We start by looking at the natural numbers:

```
instance GEnum Nat where
  genum = map to genumRep
```

Note that instead of using the default definition we directly inline its definition; this is to circumvent a bug in the current implementation of defaults that prevents later rewrite rules from applying. GHC then generates the following code:

$$\$x_2 :: [U_1 :+: K_1\ Nat]$$
$$\$x_2 = map\ \$x_4\ \$GEnumNat_{genum}$$
$$\$x_1 :: [U_1 :+: K_1\ Nat]$$
$$\$x_1 = \$x_3 \mathbin{|||} \$x_2$$
$$\$GEnumNat_{genum} :: [Nat]$$
$$\$GEnumNat_{genum} = map\ to\ \$x_1$$

We omit the definitions of $\$x_3$ and $\$x_4$ for brevity. To make progress we need to tell GHC to move the *map to* expression in $\$GEnumNat_{genum}$ through the $(|||)$ operator. We use a rewrite rule for this:

$$\{-\#\ RULES\ \texttt{"ft/|||"}\ \forall f\ a\ b.\ map\ f\ (a \mathbin{|||} b) = map\ f\ a \mathbin{|||} map\ f\ b\ \#-\}$$

With this rule in place, GHC generates the following code:

$$\$x_2 :: [U_1 :+: K_1\ Nat]$$
$$\$x_2 = map\ \$x_4\ \$GEnumNat_{genum}$$
$$\$x_1 :: [Nat]$$
$$\$x_1 = map\ to\ \$x_2$$

[4] The type of *eqA* is *GEq* $\alpha$, but we use it as if it had type $\alpha \to \alpha \to Bool$. In the generated core there is also a coercion around the use of *eqA* to transform the class type into a function, but we elide these details as they are not relevant to the optimisation itself.

$$\$GEnumNat_{genum} :: [Nat]$$
$$\$GEnumNat_{genum} = \$x_3 \,|||\, \$x_1$$

We now see that the $\$x_1$ term is *map* applied to the result of a *map*. The way *map* is optimised in GHC (by conversion to *build*/*foldr* form) interferes with our `"ft/|||"` rewrite rule, and map fusion is not happening. We can remedy this with an explicit map fusion rewrite rule:

$$\{-\#\ RULES\ \texttt{"map/map"}\ \forall f\ g\ l.\ map\ f\ (map\ g\ l) = map\ (f \circ g)\ l\ \#-\}$$

This rule results in much improved generated code:

$$\$x_3 :: [U_1 :+: K_1\ Nat]$$
$$\$x_3 = \$x_4 : []$$
$$\$x_2 :: [Nat]$$
$$\$x_2 = map\ to\ \$x_3$$
$$\$x_1 :: [Nat]$$
$$\$x_1 = map\ Su\ \$GEnumNat_{genum}$$
$$\$GEnumNat_{genum} :: [Nat]$$
$$\$GEnumNat_{genum} = \$x_2 \,|||\, \$x_1$$

The only thing we are missing now is to optimise $\$x_3$; note that its type is $[U_1 :+: K_1\ Nat]$, and not $[Nat]$. For this we simply need to tell GHC to eagerly map a function over a list with a single element:

$$\{-\#\ RULES\ \texttt{"map/singleton"}\ \forall f\ x.\ map\ f\ (x : []) = (f\ x) : []\ \#-\}$$

With this, GHC can finally generate the fully specialised enumeration function on *Nat*:

$$\$x_2 :: [Nat]$$
$$\$x_2 = Ze : []$$
$$\$x_1 :: [Nat]$$
$$\$x_1 = map\ Su\ \$GEnumNat_{genum}$$
$$\$GEnumNat_{genum} :: [Nat]$$
$$\$GEnumNat_{genum} = \$x_2 \,|||\, \$x_1$$

Compelling GHC to optimise generic enumeration for lists proves to be more difficult.[5] Since lists use products, we need to introduce a rewrite rule for the free theorem of *diag*, allowing *map* to be pushed inside *diag*:

$$\{-\#\ RULES\ \texttt{"ft/diag"}\ \forall f\ l.\ map\ f\ (diag\ l) = diag\ (map\ (map\ f)\ l)\ \#-\}$$

With this rule, and the extra optimisation flag `-fno-full-laziness` to maximise the chances for rewrite rules to apply, we get the following code:

$$\$GEnum[]_{genum} :: \forall \alpha.GEnum\ \alpha \Rightarrow [[\alpha]]$$
$$\$GEnum[]_{genum} = \lambda\,(gEnumA :: GEnum\ \alpha) \rightarrow$$
$$\quad ([] : []) \,|||\, \mathbf{let}\ \$x_1 :: [K_1\ [\alpha]]$$

[5] We believe, however, that this is only due to bugs in the inliner, and have filed bug reports #7109, #7112, and #7114 to address these issues.

$$
\begin{array}{l}
\quad \$x_1 = map\ K_1\ (\$GEnum[]_{genum}\ gEnumA) \\
\textbf{in}\ diag\ (map\ (\lambda(\$x_3 :: \alpha) \rightarrow \\
\qquad\qquad map\ (\lambda(\$x_2 :: K_1\ [\alpha]) \rightarrow \textbf{case}\ \$x_2\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad K_1\ \$x_4 \rightarrow \$x_3 : \$x_4)\ \$x_1) \\
\qquad\qquad\quad gEnumA)
\end{array}
$$

Most of the generic overhead is optimised away, but one problem remains: $\$x_1$ maps $K_1$ over the recursive enumeration elements, but this $K_1$ is immediately eliminated by a **case** statement. If $\$x_1$ was inlined, GHC could perform a map fusion, and then eliminate the use of $K_1$ altogether. However, we have no way to specify that $\$x_1$ should be inlined; the compiler generated it, so only the compiler can decide when to inline it. Also, we had to use the compiler flag `-fno-full-laziness` to prevent some let-floating, but the flag applies to the entire program and might have unintended side-effects.

Reflecting on our developments in this section, we have seen that:

- Convincing GHC to optimise *genum* for a simple datatype such as *Nat* requires the expected free theorem of $(|||)$. However, due to interaction between phases of application of rewrite rules, we are forced to introduce new rules for optimisation of *map*.
- Optimising *genum* for a more complicated datatype like lists requires the expected free theorem of *diag*. However, even after further tweaking of optimisation flags, we are currently unable to derive a fully optimised implementation. In any case, the partial optimisation achieved is certainly beneficial.
- More generally, we see that practical optimisation of generic functions is hard because of subtle interactions between the different optimisation mechanisms involved, such as inlining, rewrite rule application, **let** floating, **case** optimisation, etc.

These experiments have been performed with GHC version 7.4.1. We have observed that the behavior of the optimiser changes between compiler versions. In particular, some techniques which resulted in better code in some versions (e.g. the use of *SPECIALISE* pragmas) result in worse code in other versions. We are working together with GHC developers to ensure that generic code, at least for the `generic-deriving` library, is specialised adequately, guaranteeing performance equivalent to type-specific code.

## 5 Benchmarking

We have confirmed the expected runtime behaviour of our code by benchmarking it. Benchmarking is, in general, a complex task, and a lazy language imposes even more challenges on the design of a benchmark. We designed a benchmark suite that ensures easy repeatability of tests, calculating the average running time and the standard deviation for statistical analysis. It is portable across different operating systems and can easily be run with different compiler versions. To ensure reliability of the benchmark we use profiling, which gives us information about which computations last longer. For each of the tests, we ensure that at least 50% of the time is spent on the function we

want to benchmark. A top-level Haskell script 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. To ensure the improvements are effective in practice, we have not used micro-benchmarking, and instead benchmark whole programs.

We have a detailed benchmark suite over different datatypes and generic functions.[6] It is, however, useless to show most of the benchmark figures; because we have inspected the resulting core code and concluded that it is equivalent to a hand-written variant, the benchmark is only a form of "sanity-check" on the optimisation. Confirming the findings of Section 4, the benchmark finds no difference between the running times of generic versus type-specific equality. We have also benchmarked a traversal that updates the values in a tree, and a conversion to *String*; in both cases, the generic function performs as fast as the handwritten code. The techniques used to optimise these functions were exactly the same as those for generic equality, and indeed we expect this to be the case for many common generic functions.

As for enumeration, we find no overhead for the *Nat* datatype. Enumeration for a binary tree datatype runs about 1.63 times slower than a type-specific variant, probably because the optimiser fails to remove all generic representation overhead (as predicted in Section 4.3). Even with the remaining problems in optimising generic enumeration, these results are a substantial improvement over our previous optimisation efforts (Magalhães et al. 2010), and rely on techniques that are far less likely to degrade performance in other parts of the code.

## 6 Conclusion

In this paper we have looked at the problem of optimising generic functions. With their representation types and associated conversions, generic programs tend to be slower than their type-specific handwritten counterparts, and this can limit adoption of generic programming in situations where performance is important. We have picked one specific library, `generic-deriving`, and investigated the code generation for generic programs, and the necessary optimisation techniques to fully remove any overhead from the library. We concluded that the overhead can be fully removed most of the time, using only already available optimisations that apply to functional programs in general. However, due to the difficulty of managing the interaction between several different optimisations, in some cases we are not able to fully remove the overhead. We are confident, however, that this is only a matter of further tweaking of GHC's optimisation strategies, and fixing some open bugs.

### 6.1 Automatic inlining and generation of rewrite rules

Some work remains to be done in terms of improving the user experience. We have mentioned that the *to* and *from* functions should be inlined; this should be automatically

[6] `https://bitbucket.org/dreixel/public/src/7d32c569e678/benchmark`

established by the mechanism for deriving *Generic* instances. Additionally, inserting *INLINE* pragmas for each case in the generic function is a tedious process, which should also be automated. Finally, it would be interesting to see if the definition of rewrite rules based on free theorems of auxiliary functions used could be automated; it is easy to generate free theorems, but it is not always clear how to use these theorems for optimisation purposes.

### 6.2 Optimising other libraries

The library we have used for the development in this paper, `generic-deriving`, is practical, realistic, and representative of many other libraries. In particular, our techniques readily apply to `regular` (Van Noort et al. 2008) and `instant-generics` (Chakravarty et al. 2009), for instance.

Other approaches to generic programming, such as Scrap Your Boilerplate (SYB, Lämmel and Peyton Jones 2003, 2004), use different implementation mechanisms and require different optimisation strategies. SYB, in particular, cannot be optimised using the same techniques we have seen, because it relies on (type-safe) runtime casts. Since type comparisons are performed at runtime, the compiler does not have enough information to automatically specialise generic functions. It remains to be seen how to optimise other approaches, and to establish general guidelines for optimisation of generic programs.

In any case, it is now clear that generic programs do not have to be slow, and their optimisation up to handwritten code performance is not only possible but also achievable using only standard optimisation techniques. This opens the door for a future where generic programs are not only general, elegant, and concise, but also as efficient as type-specific code.

## Bibliography

Artem Alimarine and Sjaak Smetsers. Optimizing generic functions. In Dexter Kozen and Carron Shankland, editors, *Mathematics of Program Construction, 7th International Conference*, volume 3125 of *LNCS*, pages 16–31. Springer, 2004. doi: 10.1007/978-3-540-27764-4_3.

Artem Alimarine and Sjaak Smetsers. Improved fusion for optimizing generics. In Manuel V. Hermenegildo and Daniel Cabeza, editors, *Practical Aspects of Declarative Languages, 7th International Symposium*, volume 3350 of *LNCS*, pages 203–218. Springer, 2005. doi: 10.1007/978-3-540-30557-6_16.

Manuel M. T. Chakravarty, Gabriel C. Ditu, and Roman Leshchinskiy. Instant generics: Fast and easy, 2009. Available at `http://www.cse.unsw.edu.au/~chak/papers/CDL09.html`.

Ralf Hinze, Johan Jeuring, and Andres Löh. Comparing approches to generic programming in Haskell. In Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring, editors, *Datatype-Generic Programming*, volume 4719 of *LNCS*, pages 72–149. Springer-Verlag, 2007. doi: 10.1007/978-3-540-76786-2_2.

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*, pages 26–37. ACM, 2003. doi: 10.1145/604174.604179.

Ralf Lämmel and Simon Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *Proceedings of the 9th ACM SIGPLAN International Conference on Functional Programming*, pages 244–255. ACM, 2004. doi: 10.1145/1016850.1016883.

José Pedro Magalhães. *Less Is More: Generic Programming Theory and Practice*. PhD thesis, Universiteit Utrecht, 2012.

José Pedro Magalhães, Stefan Holdermans, Johan Jeuring, and Andres Löh. Optimizing generics is easy! In *Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 33–42. ACM, 2010. doi: 10.1145/1706356.1706366.

Thomas van Noort, Alexey Rodriguez Yakushev, Stefan Holdermans, Johan Jeuring, and Bastiaan Heeren. A lightweight approach to datatype-generic rewriting. In *Proceedings of the ACM SIGPLAN Workshop on Generic Programming*, pages 13–24. ACM, 2008. doi: 10.1145/1411318.1411321.

Thomas van Noort, Alexey Rodriguez Yakushev, Stefan Holdermans, Johan Jeuring, Bastiaan Heeren, and José Pedro Magalhães. A lightweight approach to datatype-generic rewriting. *Journal of Functional Programming*, 20(Special Issue 3–4):375–413, 2010. doi: 10.1017/S0956796810000183.

Simon Peyton Jones and Simon Marlow. Secrets of the Glasgow Haskell Compiler inliner. *Journal of Functional Programming*, 12(4&5):393–433, 2002. doi: 10.1017/S0956796802004331.

Simon Peyton Jones and André L. M. Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32:3–47, September 1998. doi: 10.1016/S0167-6423(97)00029-4.

Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the rules: Rewriting as a practical optimisation technique in GHC. In *Haskell Workshop 2001*, pages 203–233, 2001.

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 *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell*, pages 111–122. ACM, 2008. doi: 10.1145/1411286.1411301.

David Sands. Improvement theory and its applications. In A. D. Gordon and A. M. Pitts, editors, *Higher Order Operational Techniques in Semantics*, Publications of the Newton Institute, pages 275–306. Cambridge University Press, 1998.

Philip Wadler. Theorems for free! In *Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture*, pages 347–359. ACM, 1989. doi: 10.1145/99370.99404.

Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving Haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, pages 53–66. ACM, 2012. doi: 10.1145/2103786.2103795.