Optimisation of Generic Programs through Inlining

José Pedro Magalhães

University of Oxford

September 14, 2012

The problem

- Generic programs are often slower than type-specific variants
- Conversions to and from representation types do not get eliminated
- No one wants to pay a performance penalty to use generic programs

Outline

- An example
  - Enumeration on naturals
  - Optimisation
- Enumeration in `generic-deriving`
- Automatic optimisation
- Conclusion

Enumeration on natural numbers I

Consider a simple example, enumeration on natural numbers:

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

$$\mathit{enumNat} :: [\mathit{Nat}]$$
$$\mathit{enumNat} = [\mathit{Ze}] \mathbin{|||} \mathit{map}\ \mathit{Su}\ \mathit{enumNat}$$

$$\textbf{infixr}\ 5\ |||$$
$$(|||) :: [\alpha] \to [\alpha] \to [\alpha]$$

Enumeration on natural numbers I

Consider a simple example, enumeration on natural numbers:

```
data Nat = Ze | Su Nat

enumNat :: [Nat]
enumNat = [Ze] ||| map Su enumNat

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

Now let's simulate a generic representation of naturals:

```
type RepNat = Either () Nat

toNat :: RepNat → Nat
toNat n = case n of
            Left ()  → Ze
            Right n → Su n
```

Enumeration on natural numbers II

We now need enumeration on units and sums:

$$
\begin{aligned}
&\mathit{enumU} :: [()] \\
&\mathit{enumU} = [()] \\
&\mathit{enumPlus} :: [\alpha] \to [\beta] \to [\mathit{Either}\ \alpha\ \beta] \\
&\mathit{enumPlus}\ \mathit{ea}\ \mathit{eb} = \mathit{map}\ \mathit{Left}\ \mathit{ea} \mid\mid\mid \mathit{map}\ \mathit{Right}\ \mathit{eb}
\end{aligned}
$$

Enumeration on natural numbers II

We now need enumeration on units and sums:

$$
\begin{aligned}
&\mathit{enumU} :: [()] \\
&\mathit{enumU} = [()] \\
&\mathit{enumPlus} :: [\alpha] \to [\beta] \to [\mathit{Either}\ \alpha\ \beta] \\
&\mathit{enumPlus}\ \mathit{ea}\ \mathit{eb} = \mathit{map}\ \mathit{Left}\ \mathit{ea} \mid\mid\mid \mathit{map}\ \mathit{Right}\ \mathit{eb}
\end{aligned}
$$

With these, we can get enumeration on *RepNat*:

$$
\begin{aligned}
&\mathit{enumRepNat} :: [\mathit{RepNat}] \\
&\mathit{enumRepNat} = \mathit{enumPlus}\ \mathit{enumU}\ \mathit{enumNatFromRep} \\
&\mathit{enumNatFromRep} :: [\mathit{Nat}] \\
&\mathit{enumNatFromRep} = \mathit{map}\ \mathit{toNat}\ \mathit{enumRepNat}
\end{aligned}
$$

We now have a type-specific enumeration for naturals, *enumNat*, and a generic one, via type representations, *enumNatFromRep*.

Optimisation of enumeration I

Let's show that $enumNatFromRep \equiv enumNat$:

$$
\begin{aligned}
& map\ toNat\ enumRepNat \\
\equiv\langle\ & \text{inline } enumRepNat\ \rangle \\
& map\ toNat\ (enumPlus\ enumU\ enumNatFromRep) \\
\equiv\langle\ & \text{inline } enumPlus\ \rangle \\
& map\ toNat\ (map\ Left\ enumU\ |||\ map\ Right\ enumNatFromRep) \\
\equiv\langle\ & \text{inline } enumU\ \rangle \\
& map\ toNat\ (map\ Left\ [()]\ |||\ map\ Right\ enumNatFromRep) \\
\equiv\langle\ & \text{inline } map\ \rangle \\
& map\ toNat\ ([Left\ ()]\ |||\ map\ Right\ enumNatFromRep)
\end{aligned}
$$

Optimisation of enumeration II

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

Generic enumeration I

Things are not much different when using an actual generic programming library. We use `generic-deriving`:

$$
\begin{aligned}
&\textbf{data}\ U_1 \qquad\quad \tau = U_1\\
&\textbf{data}\ K_1\ \iota\ \gamma \quad \tau = K_1\ \gamma\\
&\textbf{data}\ (\phi :+: \psi)\ \tau = L_1\ (\phi\ \tau) \mid R_1\ (\psi\ \tau)\\
&\textbf{data}\ (\phi :\times: \psi)\ \tau = \phi\ \tau :\times: \psi\ \tau
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{class}\ \mathit{Generic}\ \alpha\ \textbf{where}\\
&\quad \textbf{type}\ \mathit{Rep}\ \alpha :: \star \to \star\\
&\quad \mathit{to}\ \ :: \mathit{Rep}\ \alpha\ \chi \to \alpha\\
&\quad \mathit{from} :: \alpha \to \mathit{Rep}\ \alpha\ \chi
\end{aligned}
$$

Generic enumeration II

We first define enumeration on the representation types:

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

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

Generic enumeration III

$$\textbf{instance}\ (\mathit{GEnumRep}\ \alpha, \mathit{GEnumRep}\ \beta) \Rightarrow \mathit{GEnumRep}\ (\alpha :+: \beta)\ \textbf{where}$$
$$\quad \mathit{genumRep} = \mathit{map}\ L_1\ \mathit{genumRep} \mid\mid\mid \mathit{map}\ R_1\ \mathit{genumRep}$$

$$\textbf{instance}\ (\mathit{GEnumRep}\ \alpha, \mathit{GEnumRep}\ \beta) \Rightarrow \mathit{GEnumRep}\ (\alpha :\times: \beta)\ \textbf{where}$$
$$\quad \mathit{genumRep} = \mathit{diag}\ (\mathit{map}\ (\lambda x \rightarrow$$
$$\qquad\qquad \mathit{map}\ (\lambda y \rightarrow x :\times: y)\ \mathit{genumRep})\ \mathit{genumRep})$$

$$\mathit{diag} :: [[\alpha]] \rightarrow [\alpha]$$
$$\mathit{diag} = \ldots$$

We need interleaving and diagonalisation operations, whose definitions we omit.

Generic enumeration IV

Now we can define enumeration on user types:

```
class GEnum α where
  genum :: [α]
  default genum :: (Generic α, GEnumRep (Rep α)) ⇒ [α]
  genum = map to genumRep
```

Ad hoc instances:

```
instance GEnum Int where
  genum = [0..] ||| map negate [1..]
```

Generic instances:

```
instance (GEnum α) ⇒ GEnum [α]
```

Practical guidelines I

Inline pragmas on generic functions:

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

instance GEnumRep U1 where
  {-# INLINE genumRep #-}
  genumRep = [U1]

instance (GEnum γ) ⇒ GEnumRep (K1 ι γ) where
  {-# INLINE genumRep #-}
  genumRep = map K1 genum

...
```

Practical guidelines II

Inline pragmas on conversion functions:

```
instance Generic Nat where
  type Rep Nat = U1 :+: Rec0 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)
```

Do not inline immediately; let the generic function be inlined first.

Core code for generic enumeration I

With the inline pragmas in place, we get the following core code for enumeration on natural numbers:

$$\$x_2 :: [\,U_1 :+: Rec_0\ Nat\,]$$
$$\$x_2 = map\ \$x_4\ \$GEnumNat_{genum}$$
$$\$x_1 :: [\,U_1 :+: Rec_0\ Nat\,]$$
$$\$x_1 = \$x_3\ |||\ \$x_2$$
$$\$GEnumNat_{genum} :: [\,Nat\,]$$
$$\$GEnumNat_{genum} = map\ to\ \$x_1$$

Core code for generic enumeration II

Let's add a rewrite rule for the free theorem of (|||):

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

We then get:

$$\$x_2 :: [U_1 :+: Rec_0\ Nat]$$
$$\$x_2 = map\ \$x_4\ \$GEnumNat_{genum}$$
$$\$x_1 :: [Nat]$$
$$\$x_1 = map\ to\ \$x_2$$
$$\$GEnumNat_{genum} :: [Nat]$$
$$\$GEnumNat_{genum} = \$x_3\ |||\ \$x_1$$

Core code for generic enumeration III

We need an explicit map fusion rewrite rule:

$$\{-\#\ \mathit{RULES}\ \texttt{"map/map1"}\ \forall f\ g\ l.\ \mathit{map}\ f\ (\mathit{map}\ g\ l) = \mathit{map}\ (f \circ g)\ l\ \#-\}$$

This rule results in much improved core code:

$$\$x_3 :: [\,U_1 :+: \mathit{Rec}_0\ \mathit{Nat}\,]$$
$$\$x_3 = \$x_4 : []$$
$$\$x_2 :: [\,\mathit{Nat}\,]$$
$$\$x_2 = \mathit{map}\ \mathit{to}\ \$x_3$$
$$\$x_1 :: [\,\mathit{Nat}\,]$$
$$\$x_1 = \mathit{map}\ \mathit{Su}\ \$\mathit{GEnumNat}_{\mathit{genum}}$$
$$\$\mathit{GEnumNat}_{\mathit{genum}} :: [\,\mathit{Nat}\,]$$
$$\$\mathit{GEnumNat}_{\mathit{genum}} = \$x_2 \,|||\, \$x_1$$

Core code for generic enumeration IV

One more rule:

$$\{-\#\ \mathit{RULES}\ \texttt{"map/map2"}\ \forall f\ x.\ \mathit{map}\ f\ (x : []) = (f\ x) : []\ \#-\}$$

And we're done:

$$\$x_2 :: [\mathit{Nat}]$$
$$\$x_2 = \mathit{Ze} : []$$
$$\$x_1 :: [\mathit{Nat}]$$
$$\$x_1 = \mathit{map}\ \mathit{Su}\ \$\mathit{GEnumNat}_{genum}$$
$$\$\mathit{GEnumNat}_{genum} :: [\mathit{Nat}]$$
$$\$\mathit{GEnumNat}_{genum} = \$x_2\ |||\ \$x_1$$

Please read the paper! :-)

Omitted from this talk:

- Another example: generic equality
- Optimise equality and enumeration for lists
- Benchmark results

Draft paper "Optimisation of Generic Programs through Inlining" on my website: dreixel.net

Conclusion and future work

- Generic functions don't have to be slow!
- GHC already has all the necessary infrastructure; we only need to tell it what to do
- Still, getting GHC to do it can be tricky...
- Different optimisations interfere with each other

Things to think about:

- We can automate the generation of inline pragmas. Can we automate the generation of rewrite rules?
- How to optimise other approaches to generic programming?