

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

- ▶ 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:

```
data Nat = Ze | Su Nat
enumNat :: [Nat]
enumNat = [Ze] ||| map Su enumNat
infix 5 |||
(|||) :: [α] → [α] → [α]
```

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
infix 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:

$$\text{enumU} :: [()]$$
$$\text{enumU} = [()]$$
$$\text{enumPlus} :: [\alpha] \rightarrow [\beta] \rightarrow [\text{Either } \alpha \beta]$$
$$\text{enumPlus } ea \ eb = \text{map Left } ea \ ||| \ \text{map Right } eb$$

Enumeration on natural numbers II



We now need enumeration on units and sums:

```
enumU :: [()]  
enumU = [()]  
enumPlus :: [α] → [β] → [Either α β]  
enumPlus ea eb = map Left ea ||| map Right eb
```

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

```
enumRepNat :: [RepNat]  
enumRepNat = enumPlus enumU enumNatFromRep  
enumNatFromRep :: [Nat]  
enumNatFromRep = map toNat enumRepNat
```

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 $enumNatFromRep \equiv enumNat$:

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

Optimisation of enumeration II



$map\ toNat\ ([Left\ ()])\ |||\ map\ Right\ enumNatFromRep$

$\equiv \langle \text{free theorem } (|||) : \forall f\ a\ b. map\ f\ (a\ |||\ b) = map\ f\ a\ |||\ map\ f\ b \rangle$
 $map\ toNat\ [Left\ ()]\ |||\ map\ toNat\ (map\ Right\ enumNatFromRep)$

$\equiv \langle \text{inline } map \rangle$
 $[toNat\ (Left\ ())]\ |||\ map\ toNat\ (map\ Right\ enumNatFromRep)$

$\equiv \langle \text{inline } toNat\ \text{ and case-of-constant} \rangle$
 $[Ze]\ |||\ map\ toNat\ (map\ Right\ enumNatFromRep)$

$\equiv \langle \text{functor composition law: } \forall f\ g\ l. map\ f\ (map\ g\ l) = map\ (f \circ g)\ l \rangle$
 $[Ze]\ |||\ map\ (toNat \circ Right)\ enumNatFromRep$

$\equiv \langle \text{inline } toNat\ \text{ and case-of-constant} \rangle$
 $[Ze]\ |||\ map\ Su\ enumNatFromRep \blacksquare$

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

```
data  $U_1$        $\tau = U_1$   
data  $K_1 \iota \gamma$   $\tau = K_1 \gamma$   
data  $(\phi :+ : \psi)$   $\tau = L_1 (\phi \tau) \mid R_1 (\psi \tau)$   
data  $(\phi : \times : \psi)$   $\tau = \phi \tau : \times : \psi \tau$ 
```

```
class Generic  $\alpha$  where  
  type Rep  $\alpha :: \star \rightarrow \star$   
  to   $:: \text{Rep } \alpha \chi \rightarrow \alpha$   
  from  $:: \alpha \rightarrow \text{Rep } \alpha \chi$ 
```

We first define enumeration on the representation types:

```
class GEnumRep  $\phi$  where  
  genumRep :: [ $\phi$   $\alpha$ ]
```

```
instance GEnumRep  $U_1$  where  
  genumRep = [ $U_1$ ]
```

```
instance (GEnumRep  $\gamma$ )  $\Rightarrow$  GEnumRep ( $K_1 \iota \gamma$ ) where  
  genumRep = map  $K_1$  genum
```

instance (*GEnumRep* α , *GEnumRep* β) \Rightarrow *GEnumRep* (α :+ : β) **where**
genumRep = *map* L_1 *genumRep* ||| *map* R_1 *genumRep*

instance (*GEnumRep* α , *GEnumRep* β) \Rightarrow *GEnumRep* (α : \times : β) **where**
genumRep = *diag* (*map* ($\lambda x \rightarrow$
 $\text{map } (\lambda y \rightarrow x \text{ :}\times\text{ : } y)$ *genumRep*) *genumRep*)

diag :: $[[\alpha]] \rightarrow [\alpha]$
diag = ...

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

Generic enumeration IV



Now we can define enumeration on user types:

```
class GEnum  $\alpha$  where  
  genum :: [ $\alpha$ ]  
  default genum :: (Generic  $\alpha$ , GEnumRep (Rep  $\alpha$ ))  $\Rightarrow$  [ $\alpha$ ]  
  genum = map to genumRep
```

Ad hoc instances:

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

Generic instances:

```
instance (GEnum  $\alpha$ )  $\Rightarrow$  GEnum [ $\alpha$ ]
```

Inline pragmas on generic functions:

```
class GEnumRep  $\phi$  where  
  genumRep :: [ $\phi$   $\alpha$ ]
```

```
instance GEnumRep  $U_1$  where  
  { $-\#$  INLINE genumRep  $\#-$ }  
  genumRep = [ $U_1$ ]
```

```
instance (GEnum  $\gamma$ )  $\Rightarrow$  GEnumRep ( $K_1$   $\iota$   $\gamma$ ) where  
  { $-\#$  INLINE genumRep  $\#-$ }  
  genumRep = map  $K_1$  genum
```

...

Inline pragmas on conversion functions:

instance *Generic Nat* where
type *Rep Nat* = U_1 *:+:* *Rec₀ Nat*

{-# *INLINE* [1] *to* #-}
to ($L_1 U_1$) = *Ze*
to ($R_1 (K_1 n)$) = *Su n*

{-# *INLINE* [1] *from* #-}
from *Ze* = $L_1 U_1$
from (*Su n*) = $R_1 (K_1 n)$

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

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

```
 $\$x_2 :: [U_1 :+ : \text{Rec}_0 \text{Nat}]$   
 $\$x_2 = \text{map } \$x_4 \text{ } \$GEnumNat_{genum}$   
 $\$x_1 :: [U_1 :+ : \text{Rec}_0 \text{Nat}]$   
 $\$x_1 = \$x_3 \parallel \parallel \$x_2$   
 $\$GEnumNat_{genum} :: [\text{Nat}]$   
 $\$GEnumNat_{genum} = \text{map to } \$x_1$ 
```


Core code for generic enumeration II



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

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

We then get:

$$\$x_2 :: [U_1 \text{ } \text{:+: } \text{Rec}_0 \text{ Nat}]$$

$$\$x_2 = \text{map } \$x_4 \text{ } \$GEnumNat_{genum}$$

$$\$x_1 :: [\text{Nat}]$$

$$\$x_1 = \text{map to } \$x_2$$

$$\$GEnumNat_{genum} :: [\text{Nat}]$$

$$\$GEnumNat_{genum} = \$x_3 ||| \$x_1$$

Core code for generic enumeration III



We need an explicit map fusion rewrite rule:

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

This rule results in much improved core code:

```
$x3 :: [U1 :+: Rec0 Nat]
$x3 = $x4 : []
$x2 :: [Nat]
$x2 = map to $x3
$x1 :: [Nat]
$x1 = map Su $GEnumNat_genum
$GEnumNat_genum :: [Nat]
$GEnumNat_genum = $x2 ||| $x1
```

One more rule:

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

And we're done:

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

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

- ▶ 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?