Optimizing Generics Is Easy!

José Pedro Magalhães

Joint work with
Stefan Holdermans   Johan Jeuring   Andres Löh

Dept. of Information and Computing Sciences, Universiteit Utrecht
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
Webpage: http://www.dreixel.net

FP dag 2010, Radboud Universiteit Nijmegen, 08/01/2010
Overview

Generic programming

Optimizing generics through inlining

A benchmark suite for generics

Conclusions and future work
1. Generic programming
What is datatype-generic programming?

- Programming with the *structure* of types
- Conversion functions map user datatypes to/from *representation types*
- Generic functions are defined on representation types

Generic functions work for all types for which we can write conversion functions.
Haskell supports the definition of algebraic datatypes, like:

```haskell
data Logic = Logic ∨ Logic  -- disjunction
           | Var String    -- variables
           | Not Logic     -- negation
```

To represent these, we need to know how to handle:

- Different alternatives: disjoint sums.
- Arguments of a constructor: products.
- Constructors and field labels.
- Primitive types: `String`, `Int`, ...
- ...

[Faculty of Science Information and Computing Sciences]
Generic representation II

We need to translate every datatype to this set of constructs and apply the appropriate code in the right place.

Haskell’s `data` construct combines several features: type abstraction, type recursion, (labeled) sums, and (possibly labeled) products, but they are essentially *sums of products*. 
Generic representation II

We need to translate every datatype to this set of constructs and apply the appropriate code in the right place.

Haskell’s **data** construct combines several features: type abstraction, type recursion, (labeled) sums, and (possibly labeled) products, but they are essentially *sums of products*.

To represent them we can use the following *representation datatypes*:

\[
\begin{align*}
\text{data } (f + g) & \ r = L(f \ r) \ | \ R(g \ r) & \quad \text{-- Choice} \\
\text{data } (f \times g) & \ r = f \ r \times g \ r & \quad \text{-- Multiple arguments} \\
\text{data } K & \ a \ r = K \ a & \quad \text{-- Constants} \\
\text{data } I & \ r = I \ r & \quad \text{-- Recursive occurrences} \\
\text{data } U & \ r = U & \quad \text{-- No arguments}
\end{align*}
\]
We also need to represent constructors:

\[
\begin{align*}
\text{data } & \ C \ c \ f \ r = C \ (f \ r) \\
\text{class } & \ Con \ c \ \text{where} \\
conName & :: \ t \ c \ (f :: \ast \rightarrow \ast) \ r \rightarrow \text{String}
\end{align*}
\]

We encapsulate conversion to and from the generic representation using a type class. The generic type is given using a type family:

\[
\begin{align*}
\text{type family } & \ PF \ a :: \ast \rightarrow \ast \\
\text{class } & \ Regular \ a \ \text{where} \\
from & :: \ a \rightarrow PF \ a \ a \\
to & :: PF \ a \ a ightarrow a
\end{align*}
\]
Back to our *Logic* example:

\[
\text{type instance } PF\ Logic = (I \times I) + (K\ String) + I \quad -- \text{disjunction}
\]

\[
\quad + (K\ String) \quad -- \text{variables}
\]

\[
\quad + I \quad -- \text{negation}
\]
Back to our *Logic* example:

**PF Logic**

\[
\text{type instance } PF \ Logic = (I \times I) + (K \ String) + I
\]

**Regular Logic**

\[
\text{instance Regular Logic where}
\]

\[
\begin{align*}
\text{from } (p \lor q) &= L((I\ p) \times (I\ q)) \\
\text{from } (\text{Var } x) &= R(L(K\ x)) \\
\text{from } (\text{Not } p) &= R(R(I\ p)) \\
\text{to } (L((I\ p) \times (I\ q))) &= p \lor q \\
\text{to } (R(L(K\ x))) &= \text{Var } x \\
\text{to } (R(R(I\ p)))) &= \text{Not } p
\end{align*}
\]

We omit constructor information for simplicity.
Generic functions: \( gmap \)

Now we can write generic functions:

```
class GMap f where
    gmap :: (a \to b) \to f a \to f b
```

Generic functions: \( gmap \)

Now we can write generic functions:

```haskell
class GMap f where
  gmap :: (a \rightarrow b) \rightarrow f a \rightarrow f b

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) \Rightarrow 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) \Rightarrow GMap (f \times g) where
  gmap f (x \times y) = gmap f x \times gmap f y
```
Generic functions: \textit{gshow} I

Another function we can define is generic show. For that we need to use constructor information.

\begin{verbatim}
class GShow f where
gshowf :: (a -> String) -> f a -> String
\end{verbatim}
Generic functions: \texttt{gshow I}

Another function we can define is generic show. For that we need to use constructor information.

\begin{verbatim}
class GShow f where
    gshowf :: (a → String) → f a → String

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 (Con c, GShow f) ⇒ GShow (C c f) where
    gshowf f cx@(C x) = "(" ++ conName cx ++ " "
                          ++ gshowf f x ++ ")"
\end{verbatim}
Generic functions: \textit{gshow I I}

\begin{align*}
\text{instance } & (\text{GShow } f, \text{GShow } g) \Rightarrow \text{GShow } (f + g) \text{ where } \\
& \quad gshow f (L \ x) = gshow f x \\
& \quad gshow f (R \ x) = gshow f x
\end{align*}

\begin{align*}
\text{instance } & (\text{GShow } f, \text{GShow } g) \Rightarrow \text{GShow } (f \times g) \text{ where } \\
& \quad gshow f (x \times y) = gshow f x ++ " " ++ \text{gshow} f y
\end{align*}
Generic functions: \textit{gshow} II

\begin{align*}
\text{instance } (GShow f, GShow g) & \Rightarrow GShow (f + g) \text{ where} \\
gshowf f (L \ x) & = gshowf f x \\
gshowf f (R \ x) & = gshowf f x
\end{align*}

\begin{align*}
\text{instance } (GShow f, GShow g) & \Rightarrow GShow (f \times g) \text{ where} \\
gshowf f (x \times y) & = gshowf f x + \ " \ + \ gshowf f y
\end{align*}

This function works only on the generic representations. For normal datatypes we first have to convert them:

\begin{align*}
gshow :: (\text{Regular } a, GShow (PF \ a)) & \Rightarrow a \rightarrow \text{String} \\
gshow \ x & = gshowf gshow (\text{from } x)
\end{align*}

At the recursive occurrences we apply \textit{gshow} again.
2. Optimizing generics through inlining
While representation types are useful, they incur a performance penalty:

- Generic functions keep converting back and forth
- Generic representation types are present in the final generated code
- Even “fast” generic programming libraries typically perform 2–4 times slower than handwritten variants
- “Slower” libraries can be up to 8–16 times slower
Generic representation types should not be present in the generated code. Generic functions can be specialized to particular types.

We can see that if we inline definitions and apply equational reasoning we can remove the generic representations.

As an example, let us see one-level generic identity on the `Logic` datatype:

\[
gid_{Logic} :: Logic \rightarrow Logic
\]

\[
gid_{Logic} = to \circ gmap id \circ from
\]
Efficiency III

to \((gmap \ id \ (from \ l))\)
Efficiency III

\[ \text{to} \ (gmap \ id \ (from \ l)) \]

\[ \Rightarrow \ \{ \ \text{choose} \ l \ \text{to be} \ p \lor q \ (\text{other constructors similar}) \ \} \]

\[ \text{to} \ (gmap \ id \ (from \ (p \lor q))) \]
to \((g\text{map } id \ (from \ l))\)

\[\Rightarrow \{ \text{choose } l \text{ to be } p \lor q \ (\text{other constructors similar}) \} \]

\[to \ (g\text{map } id \ (from \ (p \lor q))))\]

\[\equiv \{ \text{definition of } from_{\text{Logic}} \} \]

\[to \ (g\text{map } id \ (L \ (I \ p \times I \ q))))\]
Efficiency III

to \((gmap \text{id} \,(\text{from}\ l))\)

\[ \Rightarrow \{ \text{choose } l \text{ to be } p \lor q \text{ (other constructors similar) } \} \\
\text{to} \ (gmap \text{id} \,(\text{from} \ (p \lor q)))\]

\[ \equiv \{ \text{definition of } \text{from}_{\text{Logic}} \} \\
\text{to} \ (gmap \text{id} \ (L \,(I \,p \times I \,q)))\]

\[ \equiv \{ \text{definition of } \text{gmap}_+, \, \text{gmap}\times \} \\
\text{to} \ (L \,(gmap \text{id} \,(I \,p) \times gmap \text{id} \,(I \,q)))\]
Efficiency III

to \( (gmap\ id\ (from\ l)) \)

\[ \Rightarrow \{ \text{choose } l \text{ to be } p \lor q \text{ (other constructors similar)} \} \]

to \( (gmap\ id\ (from\ (p \lor q))) \)

\[ \equiv \{ \text{definition of } from_{Logic} \} \]

to \( (gmap\ id\ (L\ (I\ p \times I\ q))) \)

\[ \equiv \{ \text{definition of } gmap_+, gmap\times \} \]

to \( (L\ (gmap\ id\ (I\ p) \times gmap\ id\ (I\ q))) \)

\[ \equiv \{ \text{definition of } gmap_I \} \]

to \( (L\ (I\ (id\ p) \times (I\ (id\ q)))) \)
Efficiency III

\[ \text{to } (\text{gmap } \text{id } (\text{from } l)) \]

\[ \Rightarrow \{ \text{choose } l \text{ to be } p \lor q \text{ (other constructors similar) } \} \]

\[ \text{to } (\text{gmap } \text{id } (\text{from } (p \lor q))) \]

\[ \equiv \{ \text{definition of from}_{\text{Logic}} \} \]

\[ \text{to } (\text{gmap } \text{id } (L (I p \times I q))) \]

\[ \equiv \{ \text{definition of gmap}_+, \text{gmap}_\times \} \]

\[ \text{to } (L (\text{gmap } \text{id } (I p) \times \text{gmap } \text{id } (I q))) \]

\[ \equiv \{ \text{definition of gmap}_I \} \]

\[ \text{to } (L (I (\text{id } p) \times (I (\text{id } q)))) \]

\[ \equiv \{ \text{definition of id, to}_{\text{Logic}} \} \]

\[ p \lor q \]
Can we not get the compiler to do the same for us? The core code GHC generates for our example

\[ gid_{\text{Logic}} :: \text{Logic} \rightarrow \text{Logic} \]
\[ gid_{\text{Logic}} = \text{to} \circ \text{gmap id} \circ \text{from} \]

is

\[ gid_{\text{Logic}}^{01} :: \text{Logic} \rightarrow \text{Logic} \]
\[ gid_{\text{Logic}}^{01} = \lambda(x :: \text{Logic}) \rightarrow \text{to} (\text{from} x) \]

This is good, but not ideal. We also know that
\[ to_{\text{Logic}} \circ from_{\text{Logic}} \equiv \text{id}. \]
Core code II

The problem is that the compiler is conservative with *inlining*—replacing function calls with their body. We can force inlining by tweaking some flags:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Default</th>
<th>Abbr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>-funfolding-creation-threshold</td>
<td>45</td>
<td>CT</td>
</tr>
<tr>
<td>-funfolding-use-threshold</td>
<td>6</td>
<td>UT</td>
</tr>
</tbody>
</table>

Compiling with `-O2 -funfolding-use-threshold=60` produces the wanted result:

\[
gid_{\text{Logic}^{02\text{UT}60}} :: \text{Logic} \rightarrow \text{Logic}
\]

\[
gid_{\text{Logic}^{02\text{UT}60}} = \lambda (x :: \text{Logic}) \rightarrow x
\]
Core code III

For \textit{gshow}, with standard optimizations we get:

\[
gshow^{01}_{\text{Logic}} :: \text{Logic} \rightarrow \text{String} \\
gshow^{01}_{\text{Logic}} = \lambda(x :: \text{Logic}) \rightarrow \\
\text{case } (\text{from } x) \text{`cast`} (\text{sym } (\text{trans } \ldots)) \text{ of } w \{ \\
L \ y \rightarrow \ldots \\
R \ y \rightarrow \ldots \}
\]
Core code III

For $gshow$, with standard optimizations we get:

\[
\begin{align*}
gshow_{\text{Logic}}^{01} &:: \text{Logic} \to \text{String} \\
gshow_{\text{Logic}}^{01} &= \lambda(x :: \text{Logic}) \to \\
&\quad \text{case} (\text{from } x) \text{`cast` (sym (trans ...))} \text{ of } w \{ \\
&\quad \quad \begin{cases}
\text{L } y \to \ldots \\
\text{R } y \to \ldots
\end{cases}
\}
\end{align*}
\]

But we can force inlining to obtain a better result:

\[
\begin{align*}
gshow_{\text{Logic}}^{\text{CT90UT30}} &:: \text{Logic} \to \text{String} \\
gshow_{\text{Logic}}^{\text{CT90UT30}} &= \lambda(x :: \text{Logic}) \to \text{case } x \text{ of } w \{ \\
&\quad (\lor) \ p \ q \to (\text{=}++) \ldots \ gshow_{\text{Logic}}^{\text{CT90UT30}} \ p \ldots \ gshow_{\text{Logic}}^{\text{CT90UT30}} \ q \ldots \\
&\quad \text{Var } v \to (\text{=}++) \ldots \ show \ v \ldots \\
&\quad \text{Not } p \to (\text{=}++) \ldots \ gshow_{\text{Logic}}^{\text{CT90UT30}} \ p \ldots \\
&\quad \text{Const } b \to (\text{=}++) \ldots \ show \ b \ldots
\}
\end{align*}
\]
3. A benchmark suite for generics
A benchmark suite for generics: functions

To visualize the impact of increased inlining we designed a benchmark suite of generic programs. We will show two functions:

- **show** Requires constructor information, such as name and fixity.
- **update** Transform all odd `Int` values by adding one to them, or prepend all non-empty `String` values with a "y".

In our paper we present also the results for generic equality, map and read.
A benchmark suite for generics: datatypes

We use two datatypes. The *Tree* datatype is a simple labeled binary leaf tree:

```haskell
data Tree a = Bin a (Tree a) (Tree a) | Leaf
```

The *Logic* type is similar to the one we introduced before, only with more constructors:

```haskell
data Logic = Impl Logic Logic | Equiv Logic Logic | Conj Logic Logic | Disj Logic Logic | Not Logic | Var String | T | F
```
A benchmark suite for generics: libraries

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

- **emgm** Extensible and Modular Generics for the Masses. Its fundamental characteristic is to encode datatype representations through a type class.
- **syb** Scrap Your Boilerplate is a very popular library based on generic combinators and type-safe cast. It comes with GHC.
- **regular** The library described in the introduction.
- **multirec** The first approach able to express mutually recursive datatypes. Structurally similar to **regular**, but makes use of a few more advanced concepts to deal with mutual recursion.
Results: show for Tree
Results: show for Logic
Results: update for Tree
Results: update for Logic
4. Conclusions and future work
Conclusions

- Generic programs do not have to be slow
- Inlining is the way to go
- Facilities for inlining are already present in the compiler and can be reused for optimizing generics
- Both emgm and regular are fast and can be optimized to handwritten code speed with inlining
- The slowest (but most popular) generic programming library is syb
- multirec is not benefiting much from increased inlining, as opposed to the similar regular library
Future work

▶ Specifying the behavior of the inliner should be more localized: use the INLINE pragmas of the upcoming version of GHC
▶ Not all libraries benefit equally from increased inlining: why?
  ▶ Are GADTs preventing inlining in `multirec`?
  ▶ What can we do about `syb`?
▶ Investigate generic producers more thoroughly