Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

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.

Generic representation I

Haskell supports the definition of algebraic datatypes, like:

$$
\begin{array}{llll}
\mathbf{data}\ \mathit{Logic} = & \mathit{Logic} \vee \mathit{Logic} & \text{-- disjunction} \\
\mid & \mathit{Var}\ \mathit{String} & \text{-- variables} \\
\mid & \mathit{Not}\ \mathit{Logic} & \text{-- negation}
\end{array}
$$

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*, ...
- ...

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*:

$$
\begin{array}{llll}
\mathbf{data}\ (f + g) & r = L\ (f\ r) \mid R\ (g\ r) & & \text{-- Choice} \\
\mathbf{data}\ (f \times g) & r = f\ r \times g\ r & & \text{-- Multiple arguments} \\
\mathbf{data}\ K\ a & r = K\ a & & \text{-- Constants} \\
\mathbf{data}\ I & r = I\ r & & \text{-- Recursive occurrences} \\
\mathbf{data}\ U & r = U & & \text{-- No arguments}
\end{array}
$$

Generic representation III

We also need to represent constructors:

$$\mathbf{data}\ C\ c\ f\ r = C\ (f\ r)$$

$$\begin{array}{l}\mathbf{class}\ Con\ c\ \mathbf{where} \\ \quad conName :: t\ c\ (f :: * \to *)\ r \to String\end{array}$$

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

$$\mathbf{type\ family}\ PF\ a :: * \to *$$

$$\begin{array}{l}\mathbf{class}\ Regular\ a\ \mathbf{where} \\ \quad from :: a \to PF\ a\ a \\ \quad to \ \ \ \ :: PF\ a\ a \to a\end{array}$$

Generic representation IV

Back to our *Logic* example:

$$
\begin{aligned}
\textbf{type instance } PF\ Logic = \quad & (I \times I) && \text{-- disjunction} \\
+\ & (K\ String) && \text{-- variables} \\
+\ & I && \text{-- negation}
\end{aligned}
$$

Generic representation IV

Back to our *Logic* example:

$$
\begin{aligned}
&\textbf{type instance } \mathit{PF}\ \mathit{Logic} = && (I \times I) && \text{-- disjunction}\\
& && + (K\ \mathit{String}) && \text{-- variables}\\
& && + I && \text{-- negation}
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{instance } \mathit{Regular}\ \mathit{Logic} \textbf{ where}\\
&\quad \mathit{from}\ (p \lor q) = L\ ((I\ p) \times (I\ q))\\
&\quad \mathit{from}\ (\mathit{Var}\ x) = R\ (L\ (K\ x))\\
&\quad \mathit{from}\ (\mathit{Not}\ p) = R\ (R\ (I\ p))\\
&\quad \mathit{to}\ (L\ ((I\ p) \times (I\ q))) = p \lor q\\
&\quad \mathit{to}\ (R\ (L\ (K\ x))) = \mathit{Var}\ x\\
&\quad \mathit{to}\ (R\ (R\ (I\ p))) = \mathit{Not}\ p
\end{aligned}
$$

We omit constructor information for simplicity.

Generic functions: *gmap*

Now we can write generic functions:

$$\textbf{class}\ GMap\ f\ \textbf{where}$$
$$\quad gmap :: (a \to b) \to f\ a \to f\ b$$

Generic functions: *gmap*

Now we can write generic functions:

$$
\begin{array}{l}
\textbf{class } GMap\ f \textbf{ where} \\
\quad gmap :: (a \to b) \to f\ a \to f\ b \\
\\
\textbf{instance } GMap\ I \textbf{ where} \\
\quad gmap\ f\ (I\ r) \quad\ \ = I\ (f\ r) \\
\textbf{instance } GMap\ (K\ a) \textbf{ where} \\
\quad gmap\ \_\ (K\ x) \quad = K\ x \\
\textbf{instance } GMap\ U \textbf{ where} \\
\quad gmap\ \_\ U \qquad\quad = U \\
\textbf{instance } (GMap\ f, GMap\ g) \Rightarrow GMap\ (f + g) \textbf{ where} \\
\quad gmap\ f\ (L\ x) \quad\ = L\ (gmap\ f\ x) \\
\quad gmap\ f\ (R\ x) \quad\ = R\ (gmap\ f\ x) \\
\textbf{instance } (GMap\ f, GMap\ g) \Rightarrow GMap\ (f \times g) \textbf{ where} \\
\quad gmap\ f\ (x \times y) = gmap\ f\ x \times gmap\ f\ y
\end{array}
$$

Generic functions: $gshow$ I

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

$$
\begin{array}{l}
\mathbf{class}\ \mathit{GShow}\ f\ \mathbf{where} \\
\quad \mathit{gshowf} :: (a \to \mathit{String}) \to f\ a \to \mathit{String}
\end{array}
$$

Generic functions: *gshow* I

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

$$
\begin{array}{l}
\mathbf{class}\ GShow\ f\ \mathbf{where} \\
\quad gshowf :: (a \to String) \to f\ a \to String \\
\\
\mathbf{instance}\ GShow\ I\ \mathbf{where} \\
\quad gshowf\ f\ (I\ r) \qquad = f\ r \\
\mathbf{instance}\ (Show\ a) \Rightarrow GShow\ (K\ a)\ \mathbf{where} \\
\quad gshowf\ \_\ (K\ x) \qquad = show\ x \\
\mathbf{instance}\ GShow\ U\ \mathbf{where} \\
\quad gshowf\ \_\ U \qquad\quad = \texttt{""} \\
\mathbf{instance}\ (Con\ c, GShow\ f) \Rightarrow GShow\ (C\ c\ f)\ \mathbf{where} \\
\quad gshowf\ f\ cx@(C\ x) = \texttt{"("} +\!\!+\ conName\ cx +\!\!+ \texttt{" "} \\
\qquad\qquad\qquad\qquad\quad +\!\!+\ gshowf\ f\ x +\!\!+ \texttt{")"}
\end{array}
$$

Generic functions: *gshow* II

$$
\begin{aligned}
&\textbf{instance}\ (\mathit{GShow}\ f, \mathit{GShow}\ g) \Rightarrow \mathit{GShow}\ (f + g)\ \textbf{where} \\
&\quad \mathit{gshowf}\ f\ (L\ x) \quad = \mathit{gshowf}\ f\ x \\
&\quad \mathit{gshowf}\ f\ (R\ x) \quad = \mathit{gshowf}\ f\ x
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{instance}\ (\mathit{GShow}\ f, \mathit{GShow}\ g) \Rightarrow \mathit{GShow}\ (f \times g)\ \textbf{where} \\
&\quad \mathit{gshowf}\ f\ (x \times y) = \mathit{gshowf}\ f\ x + \!\!+ \texttt{"\ \ "} + \!\!+ \mathit{gshowf}\ f\ y
\end{aligned}
$$

Generic functions: *gshow* II

$$\textbf{instance}\ (GShow\ f, GShow\ g) \Rightarrow GShow\ (f + g)\ \textbf{where}$$
$$\begin{aligned} gshowf\ f\ (L\ x) &= gshowf\ f\ x \\ gshowf\ f\ (R\ x) &= gshowf\ f\ x \end{aligned}$$

$$\textbf{instance}\ (GShow\ f, GShow\ g) \Rightarrow GShow\ (f \times g)\ \textbf{where}$$
$$gshowf\ f\ (x \times y) = gshowf\ f\ x +\!\!+ \texttt{"\ \ "} +\!\!+ gshowf\ f\ y$$

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

$$gshow :: (Regular\ a, GShow\ (PF\ a)) \Rightarrow a \rightarrow String$$
$$gshow\ x = gshowf\ gshow\ (from\ x)$$

At the recursive occurrences we apply *gshow* again.

2. Optimizing generics through inlining

Efficiency I

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

Efficiency II

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:

$$
\begin{aligned}
&gid_{Logic} :: Logic \rightarrow Logic \\
&gid_{Logic} = to \circ gmap\ id \circ from
\end{aligned}
$$

Efficiency III

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

Efficiency III

$$\mathit{to}\ (\mathit{gmap}\ \mathit{id}\ (\mathit{from}\ l))$$

$\Rightarrow$ { choose l to be $p \vee q$ (other constructors similar) }

$$\mathit{to}\ (\mathit{gmap}\ \mathit{id}\ (\mathit{from}\ (p \vee q)))$$

Efficiency III

$$\mathit{to}\ (\mathit{gmap}\ \mathit{id}\ (\mathit{from}\ l))$$

$\Rightarrow$ { choose l to be $p \vee q$ (other constructors similar) }

$$\mathit{to}\ (\mathit{gmap}\ \mathit{id}\ (\mathit{from}\ (p \vee q)))$$

$\equiv$ { definition of $\mathit{from}_{\mathit{Logic}}$ }

$$\mathit{to}\ (\mathit{gmap}\ \mathit{id}\ (L\ (I\ p \times I\ q)))$$

Efficiency III

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

Efficiency III

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

Efficiency III

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

$\Rightarrow$ { choose l to be $p \vee q$ (other constructors similar) }

$$to\ (gmap\ id\ (from\ (p \vee q)))$$

$\equiv$ { definition of $from_{Logic}$ }

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

$\equiv$ { definition of $gmap_{+}$, $gmap_{\times}$ }

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

$\equiv$ { definition of $gmap_{I}$ }

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

$\equiv$ { definition of id, to_{Logic} }

$$p \vee q$$

Core code I

Can we not get the compiler to do the same for us? The core code GHC generates for our example

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

is

$$gid^{01}_{Logic} :: Logic \rightarrow Logic$$
$$gid^{01}_{Logic} = \lambda(x :: Logic) \rightarrow to\ (from\ x)$$

This is good, but not ideal. We also know that $to_{Logic} \circ from_{Logic} \equiv id$.

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:

Flag	Default	Abbr.
`-funfolding-creation-threshold`	45	CT
`-funfolding-use-threshold`	6	UT

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

$$gid^{\mathtt{O2UT60}}_{Logic} :: Logic \rightarrow Logic$$
$$gid^{\mathtt{O2UT60}}_{Logic} = \lambda(x :: Logic) \rightarrow x$$

Core code III

For $gshow$, with standard optimizations we get:

$$
\begin{array}{l}
gshow^{\mathsf{01}}_{Logic} :: Logic \rightarrow String \\
gshow^{\mathsf{01}}_{Logic} = \lambda(x :: Logic) \rightarrow \\
\quad \textbf{case}\ (from\ x)\ \text{`}cast\text{`}\ (sym\ (trans \ldots))\ \textbf{of}\ w\ \{ \\
\qquad L\ y \rightarrow \ldots \\
\qquad R\ y \rightarrow \ldots \}
\end{array}
$$

Core code III

For $gshow$, with standard optimizations we get:

$$\begin{aligned}
&gshow^{\mathtt{01}}_{Logic} :: Logic \rightarrow String \\
&gshow^{\mathtt{01}}_{Logic} = \lambda(x :: Logic) \rightarrow \\
&\quad \textbf{case}\ (from\ x)\ \text{`}cast\text{`}\ (sym\ (trans \ldots))\ \textbf{of}\ w\ \{ \\
&\qquad L\ y \rightarrow \ldots \\
&\qquad R\ y \rightarrow \ldots \}
\end{aligned}$$

But we can force inlining to obtain a better result:

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

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:

$$\mathbf{data}\ \mathit{Tree}\ a = \mathit{Bin}\ a\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a) \mid \mathit{Leaf}$$

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

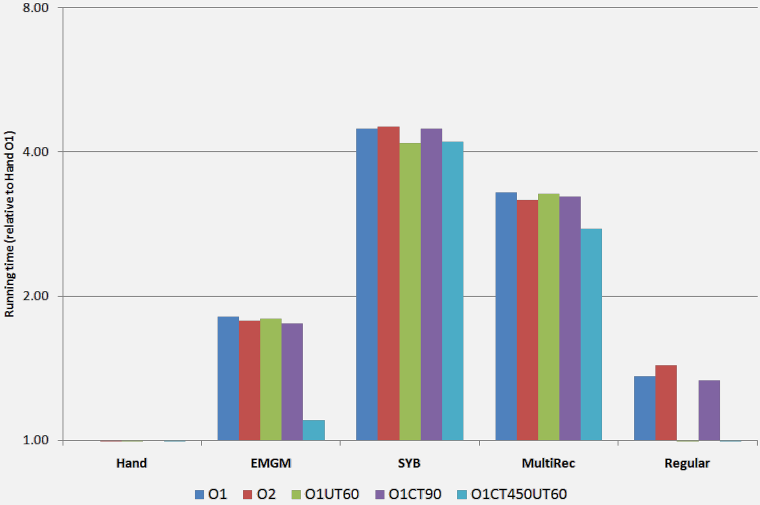
$$\begin{aligned}\mathbf{data}\ \mathit{Logic} = {} & \mathit{Impl}\ \mathit{Logic}\ \mathit{Logic} \mid \mathit{Equiv}\ \mathit{Logic}\ \mathit{Logic} \\ \mid {} & \mathit{Conj}\ \mathit{Logic}\ \mathit{Logic} \mid \mathit{Disj}\ \mathit{Logic}\ \mathit{Logic} \\ \mid {} & \mathit{Not}\ \mathit{Logic} \mid \mathit{Var}\ \mathit{String} \mid T \mid F\end{aligned}$$

A benchmark suite for generics: libraries

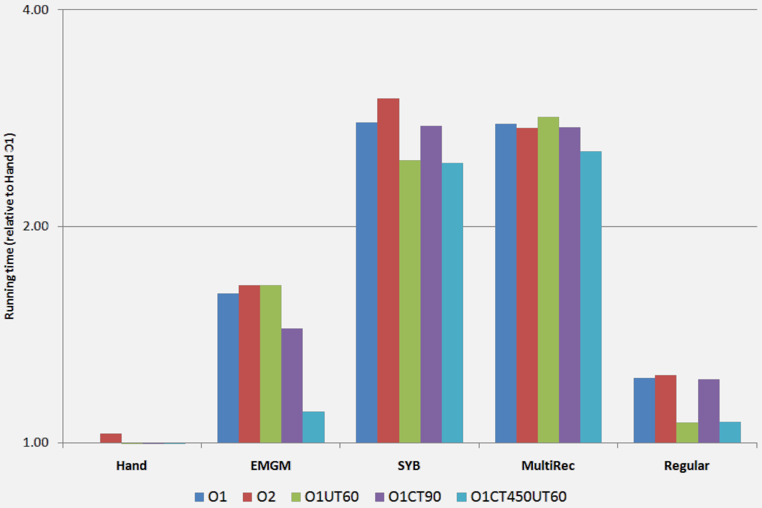
We have chosen a few representative, mainstream, and maintained libraries to benchmark:

- `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.

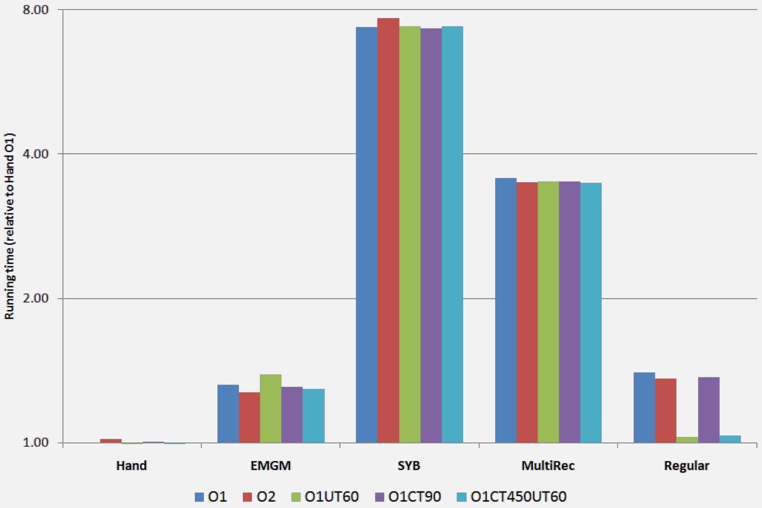
Results: *show* for *Tree*



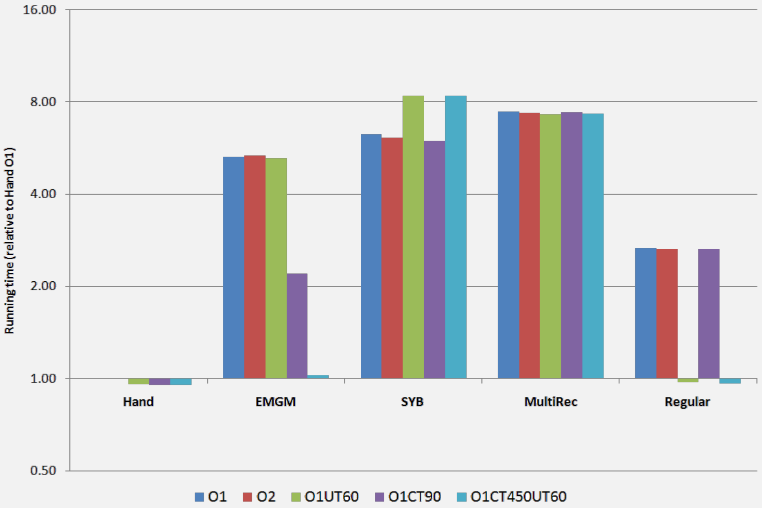
Results: *show* for *Logic*



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