Libraries for Generic Programming in Haskell

Johan Jeuring
Utrecht University and Open University, NL
johanj@cs.uu.nl

What is generic programming?

- Software development often consists of designing a datatype, to which functionality is added.
- Some functionality is datatype specific, other functionality is defined on almost all datatypes, and only depends on the *structure* of the datatype.
- This is called *datatype-generic* functionality.
- Examples of datatype-generic functionality are:
  - comparing two values for equality,
  - searching a value of a datatype for occurrences of a particular string or other value,
  - editing a value,
  - pretty-printing a value, etc.

  Larger examples include XML tools, testing frameworks, debuggers, and data-conversion tools.

Why Generic Programming?

Generic Programming is a programming technique that

- reduces code duplication
- reduces number of programming errors
- reduces software production time
- makes it easier to evolve programs

Why Generic Programming?

Generic Programming is a programming technique that

- reduces code duplication
- reduces number of programming errors
- reduces software production time
- makes it easier to evolve programs

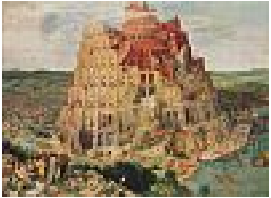
So why isn't everybody using generic programming?

Why *not* use generic programming

- generic programming tools are basically only available for Haskell
- have to download, install and use external tools or libraries
- quite a number of tools are not supported anymore
- if you want to write your own generic function: steep learning curve
- how do I choose between the remaining ten or so approaches?

Why *not* use generic programming

- generic programming tools are basically only available for Haskell
- have to download, install and use external tools or libraries
- quite a number of tools are not supported anymore
- if you want to write your own generic function: steep learning curve
- how do I choose between the remaining ten or so approaches?



Ultimate goal

We propose to design a *common generic programming library* for Haskell, for which we will guarantee continuing support.

To ensure continuing support, we will develop this library in an international committee.

Why a library?

- Haskell is powerful enough to support most generic programming concepts by means of a library.
- Compared with a language extension (PolyP, Generic Haskell), a library is much easier to ship, support, and maintain.
- Compared with a preprocessing tool like DrIFT, or Template Haskell, a library gives you much more support, such as types.

Of course the library might be accompanied by tools.

The library should support the most common generic programming scenarios.

This talk

Before we design a new library, we want to perform an extensive evaluation of the existing libraries.

This talk

- briefly introduces one of the different libraries for generic programming, by means of example
- discusses design criteria, and criteria for evaluating generic programming libraries
- shows some interesting aspects of the evaluations we have performed so far.

Generic programming libraries in Haskell

- Lightweight Implementation of Generics and Dynamics (LIGD) (2002)

Generic programming libraries in Haskell

- Lightweight Implementation of Generics and Dynamics (LIGD) (2002)
- Strafunski (2002)

Generic programming libraries in Haskell

- Lightweight Implementation of Generics and Dynamics (LIGD) (2002)
- Strafunski (2002)
- Scrap Your Boilerplate (SYB) (2003,2004,2005)

Generic programming libraries in Haskell

- Lightweight Implementation of Generics and Dynamics (LIGD) (2002)
- Strafunski (2002)
- Scrap Your Boilerplate (SYB) (2003,2004,2005)
- Polytypic Programming in Haskell (2003)

Generic programming libraries in Haskell

- Lightweight Implementation of Generics and Dynamics (LIGD) (2002)
- Strafunski (2002)
- Scrap Your Boilerplate (SYB) (2003,2004,2005)
- Polytypic Programming in Haskell (2003)
- Generics for the Masses (2004)

Generic programming libraries in Haskell

- Lightweight Implementation of Generics and Dynamics (LIGD) (2002)
- Strafunski (2002)
- Scrap Your Boilerplate (SYB) (2003,2004,2005)
- Polytypic Programming in Haskell (2003)
- Generics for the Masses (2004)
- SYB Reloaded, Revolutions (2006)

Generic programming libraries in Haskell

- Lightweight Implementation of Generics and Dynamics (LIGD) (2002)
- Strafunski (2002)
- Scrap Your Boilerplate (SYB) (2003,2004,2005)
- Polytypic Programming in Haskell (2003)
- Generics for the Masses (2004)
- SYB Reloaded, Revolutions (2006)
- Generic programming, now! (2006)

Generic programming libraries in Haskell

- Lightweight Implementation of Generics and Dynamics (LIGD) (2002)
- Strafunski (2002)
- Scrap Your Boilerplate (SYB) (2003,2004,2005)
- Polytypic Programming in Haskell (2003)
- Generics for the Masses (2004)
- SYB Reloaded, Revolutions (2006)
- Generic programming, now! (2006)
- RepLib (2006)

Generic programming libraries in Haskell

- Lightweight Implementation of Generics and Dynamics (LIGD) (2002)
- Strafunski (2002)
- Scrap Your Boilerplate (SYB) (2003,2004,2005)
- Polytypic Programming in Haskell (2003)
- Generics for the Masses (2004)
- SYB Reloaded, Revolutions (2006)
- Generic programming, now! (2006)
- RepLib (2006)
- Smash your boilerplate (2006)

Generic programming libraries in Haskell

- Lightweight Implementation of Generics and Dynamics (LIGD) (2002)
- Strafunski (2002)
- Scrap Your Boilerplate (SYB) (2003,2004,2005)
- Polytypic Programming in Haskell (2003)
- Generics for the Masses (2004)
- SYB Reloaded, Revolutions (2006)
- Generic programming, now! (2006)
- RepLib (2006)
- Smash your boilerplate (2006)
- Almost compositional functions (2006)

Generic programming libraries in Haskell

- Lightweight Implementation of Generics and Dynamics (LIGD) (2002)
- Strafunski (2002)
- Scrap Your Boilerplate (SYB) (2003,2004,2005)
- Polytypic Programming in Haskell (2003)
- Generics for the Masses (2004)
- SYB Reloaded, Revolutions (2006)
- Generic programming, now! (2006)
- RepLib (2006)
- Smash your boilerplate (2006)
- Almost compositional functions (2006)
- Extensible and Modular Generics for the Masses (EMGM) (2006)

Generic programming libraries in Haskell

- Lightweight Implementation of Generics and Dynamics (LIGD) (2002)
- Strafunski (2002)
- Scrap Your Boilerplate (SYB) (2003,2004,2005)
- Polytypic Programming in Haskell (2003)
- Generics for the Masses (2004)
- SYB Reloaded, Revolutions (2006)
- Generic programming, now! (2006)
- RepLib (2006)
- Smash your boilerplate (2006)
- Almost compositional functions (2006)
- Extensible and Modular Generics for the Masses (EMGM) (2006)
- Uniplate (2007)

High-level design decisions

There are many ways to categorize the different libraries, but important high-level decisions are:

- Representations of types are passed explicitly to generic functions: LIGD, Replib, Generic Programming, now!
- Use generic traversals and/or typecasts: Strafunski, SYB, Uniplate.
- Use the class system to define generic functions: Generics for the Masses.

Lightweight Generics and Dynamics

- Lightweight Implementation of Generics and Dynamics (LIGD) is an approach to embedding generic functions and dynamic values into Haskell 98 augmented with existential types
- The basic idea of LIGD is to reflect the type argument onto the value level so that the typecase can be implemented by ordinary pattern matching.
- Developed by Cheney and Hinze.

Structure types

Types are reflected on the value level by means of structure types. There are structure types for units, sums, and products.

$$\begin{aligned}
&\textbf{data}\ \mathrm{Unit} &&= \mathit{Unit}\\
&\textbf{data}\ \mathrm{Sum}\ a\ b &&= \mathit{Inl}\ a \mid \mathit{Inr}\ b\\
&\textbf{data}\ \mathrm{Prod}\ a\ b &&= a \times b
\end{aligned}$$

Representing lists

$$
\begin{aligned}
&\textbf{data}\ [a] = [\,] \mid a : [a] \\
&\mathit{rList} && :: \mathrm{Rep}\ a \to \mathrm{Rep}\ [a] \\
&\mathit{rList}\ \mathit{rA} && = \mathit{RType}\ (\mathit{rSum}\ (\mathit{RCon}\ \texttt{"[]"}\ \mathit{rUnit}) \\
& && \qquad\qquad\qquad (\mathit{RCon}\ \texttt{":"}\ (\mathit{rPair}\ \mathit{rA}\ (\mathit{rList}\ \mathit{rA})))) \\
& && \qquad\qquad (\mathit{EP}\ \mathit{fromList}\ \mathit{toList})
\end{aligned}
$$

$$\textbf{data}\ \mathrm{EP}\ a\ b = \mathit{EP}\{\mathit{from} :: a \to b, \mathit{to} :: b \to a\}$$

$$
\begin{aligned}
&\mathit{fromList} && :: [a] \to \mathrm{Sum\ Unit}\ (\mathrm{Prod}\ a\ [a]) \\
&\mathit{fromList}\ [\,] && = \mathit{Inl\ Unit} \\
&\mathit{fromList}\ (x : \mathit{xs}) && = \mathit{Inr}\ (x \times \mathit{xs}) \\
&\mathit{toList} && :: \mathrm{Sum\ Unit}\ (\mathrm{Prod}\ a\ [a]) \to [a] \\
&\mathit{toList}\quad (\mathit{Inl\ Unit}) && = [\,] \\
&\mathit{toList}\quad (\mathit{Inr}\ (x \times \mathit{xs})) && = x : \mathit{xs}
\end{aligned}
$$

Representing types

LIGD uses a parametric type for type representations: Rep t is the type representation of t.

$$
\begin{array}{llllll}
\textbf{data}\ \text{Rep}\ t = & & \mathit{RUnit} & & & (\text{EP}\ t\ \text{Unit}) \\
\mid & & \mathit{RInt} & & & (\text{EP}\ t\ \text{Int}) \\
\mid & \forall a\ b. & \mathit{RSum} & (\text{Rep}\ a) & (\text{Rep}\ b) & (\text{EP}\ t\ (\text{Sum}\ a\ b)) \\
\mid & \forall a\ b. & \mathit{RPair} & (\text{Rep}\ a) & (\text{Rep}\ b) & (\text{EP}\ t\ (\text{Prod}\ a\ b)) \\
\mid & \forall a. & \mathit{RType} & & (\text{Rep}\ a) & (\text{EP}\ t\ a) \\
\mid & & \mathit{RCon} & \text{String} & (\text{Rep}\ t) &
\end{array}
$$

Constructing structure types

$$
\begin{aligned}
&\mathit{self} &&:: \text{EP}\ a\ a\\
&\mathit{self} &&= \mathit{EP}\{\mathit{from} = \mathit{id}, \mathit{to} = \mathit{id}\}\\
&\mathit{rUnit} &&:: \text{Rep Unit}\\
&\mathit{rUnit} &&= \mathit{RUnit\ self}\\
&\mathit{rSum} &&:: \text{Rep}\ a \rightarrow \text{Rep}\ b \rightarrow \text{Rep}\ (\text{Sum}\ a\ b)\\
&\mathit{rSum\ rA\ rB} &&= \mathit{RSum\ rA\ rB\ self}
\end{aligned}
$$

Equality

Equality is the classic generic programming example.

$$
\begin{array}{llll}
\mathit{equalString} & & & :: \mathrm{String} \rightarrow \mathrm{String} \rightarrow \mathrm{Bool} \\
\mathit{equalString} & [\,] & [\,] & = \mathit{True} \\
\mathit{equalString} & [\,] & \_ & = \mathit{False} \\
\mathit{equalString} & \_ & [\,] & = \mathit{False} \\
\mathit{equalString} & (c : s) & (c' : s') & = c \equiv c' \wedge \mathit{equalString}\ s\ s'
\end{array}
$$

The algorithm is simple:

- Check whether two values are in the same alternative.
- If not, they are not equal.
- Otherwise, they are equal if all arguments are equal.

Equality on lists in LIGD

> $> geq\ (rList\ rInt)\ [1,2,3]\ [1,2,4]$
> *False*

Generic equality in LIGD

$$
\begin{array}{l}
geq :: \mathrm{Rep}\ t \to t \to t \to \mathrm{Bool} \\
geq\ (RUnit\ ep)\ t_1\ t_2 = \mathbf{case}\ (from\ ep\ t_1, from\ ep\ t_2)\ \mathbf{of} \\
\quad (Unit, Unit) \to True \\
geq\ (RInt\ ep)\ t_1\ t_2 = from\ ep\ t_1 \equiv from\ ep\ t_2 \\
geq\ (RSum\ rA\ rB\ ep)\ t_1\ t_2 = \mathbf{case}\ (from\ ep\ t_1, from\ ep\ t_2)\ \mathbf{of} \\
\quad (Inl\ a_1, Inl\ a_2) \to geq\ rA\ a_1\ a_2 \\
\quad (Inr\ b_1, Inr\ b_2) \to geq\ rB\ b_1\ b_2 \\
\quad \_ \to False \\
geq\ (RPair\ rA\ rB\ ep)\ t_1\ t_2 = \mathbf{case}\ (from\ ep\ t_1, from\ ep\ t_2)\ \mathbf{of} \\
\quad (a_1 \times b_1, a_2 \times b_2) \to \\
\qquad geq\ rA\ a_1\ a_2 \wedge geq\ rB\ b_1\ b_2 \\
geq\ (RType\ rA\ ep)\ t_1\ t_2 = geq\ rA\ (from\ ep\ t_1)\ (from\ ep\ t_2) \\
geq\ (RCon\ s\ rA)\ t_1\ t_2 = geq\ rA\ t_1\ t_2
\end{array}
$$

Evaluating the libraries

We evaluate existing libraries by means of a set of criteria.

Papers about generic programming usually give desirable criteria for generic programs. Examples of such criteria are:

- can a generic function be extended with special behaviour on a particular datatype,
- are generic functions first-class.

We develop a set of criteria based on our own ideas about generic programming, and ideas from papers about generic programming.

We have collected a set of generic functions for testing the criteria. We try to implement all of these functions in the different approaches.

The criteria: library design choices

- **Extensionality versus intensionality.** Is the selection of a generic function case done at compile time (extensional approach) or at runtime (intensional approach)?
  LIGD, SYB: intensional, EMGM: ...
- **Type representation.** How are types represented at runtime in intensional approaches? Are these representations handled explicitly (as arguments that can be pattern matched) or implicitly (as type class contexts)?
  LIGD: explicit, SYB: implicit.
- **Generic function encoding.** How are generic functions encoded? Are they Haskell functions or type class methods?
  LIGD, SYB: functions, EMGM: type class methods.

The criteria: types

- **Full reflexivity.** Different approaches allow different sets of datatypes in the domain of generic functions.
  EMGM and SYB do not allow higher-order kinded datatypes.
- **Views.** Does the library support more than one view?
  All libraries have to be reimplemented completely to support a new view. (SYB Revolutions can be viewed as a Boilerplate view of LIGD.)
- **Type universes.** Can you define a generic function on a particular set of datatypes?
  You would have to reimplement EMGM and LIGD. Don't know about SYB.
- **Intuition behind types.** Are the types as you expect them?
  EMGM, LIGD: yes. SYB: more or less.
- **Multiple type arguments.** Can a function be generic in more than one type argument?
  EMGM, LIGD, SYB: yes.

The criteria: expressiveness I

- **First-class generic functions.** Can a generic function take a generic function as an argument?
  Yes.
- **Generic functions of different arity.** The equality function can usually be defined in an approach to generic programming, but a generalisation of the function map on lists to arbitrary container types cannot be defined in all proposals.
  This is a problem for all library approaches.
- **Local redefinitions.** Can the programmer provide a custom function definition for the argument of a generic function used on a type constructor?
  After reimplementation.
- **Extensibility.** Can the programmer extend the definition of a generic function in a different module without the need for recompilation?
  LIGD, SYB: no. EMGM: to some extent.

The criteria: expressiveness II

- **Ad-hoc definitions for datatypes.** Can a generic function contain specific behaviour for a particular datatype, and let the remaining datatypes be handled generically?
  SYB, EMGM: yes. LIGD: no.
- **Ad-hoc definitions for constructors.** Can we give an ad-hoc definition for a particular constructor, and let the remaining constructors be handled generically?
  SYB, EMGM: yes, LIGD: no.
- **Properties of generic functions.** Is the approach based on a theory for generic functions?
  LIGD, EMGM: yes, SYB: no.
- **Consumers, transformers and producers.** Is the approach capable of defining consumer ($a \to T$), transformer ($a \to a$ or $a \to a'$) and producer ($T \to a$) generic functions?
  Yes.

The criteria: usability I

- **Performance.**

```
RepLib:      710;
PolyP:      1199;
EMGM:       1843;
LIGD:       2365;
Spine:      3364;
NOW:        3645;
SYB1_2:     7479;
```

The criteria: usability II

- **Portability.** LIGD, EMGM: portable (some rank-*n* polymorphism). SYB: GHC.
- **Amount of work per datatype.** SYB: none, LIGD, EMGM: structure types.
- **Error messages.** SYB: ..., LIGD, EMGM: OK.
- **Practical aspects.** SYB: well-developed, comes with GHC. LIGD: dead? EMGM: no website.
- **Ease of learning.** LIGD: easy, EMGM: intermediate, SYB: difficult.

Conclusions

- We are trying to design a common generic programming library.
- For that purpose we have evaluated exisiting libraries for generic programming in Haskell.
- I have introduced LIGD
- and criteria we would like an approach to satisfy.
- I've discussed some results of the evaluation.

More hopefully soon in a paper about our comparison!

But first: Hello World!