Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

# Generic Programming: what, why and how

José Pedro Magalhães

5th Dutch Haskell Users' Group meeting
11/09/2009

# What kind of generic?

In many languages, the function below is generic:

$$\begin{aligned} &length :: [a] \to \mathsf{Int} \\ &length\ [\,] \qquad = 0 \\ &length\ (\_ : t) = 1 + length\ t \end{aligned}$$

In Haskell, however, we call *length* a polymorphic function, and reserve the term generic for something else. . .

# Exercise assistants: Logic I

Imagine you are writing software for helping students turn logic expressions into disjunctive normal form.

# Exercise assistants: Logic I

Imagine you are writing software for helping students turn logic expressions into disjunctive normal form.

You need:

- A description of the logic domain

# Exercise assistants: Logic I

Imagine you are writing software for helping students turn logic expressions into disjunctive normal form.

You need:

- A description of the logic domain
- Functionality on that domain:
  - Parsing and pretty-printing
  - Equality and top-level equality
  - Folding
  - Exercise generation
  - ...

# Exercise assistants: Logic II

Let's get started, then:

$$
\begin{aligned}
\textbf{data}\ \mathsf{Logic} = {} & \mathsf{Logic}\ {:}{\rightarrow}{:}\ \mathsf{Logic} && \text{-- implication} \\
\mid {} & \mathsf{Logic}\ {:}{\leftrightarrow}{:}\ \mathsf{Logic} && \text{-- equivalence} \\
\mid {} & \mathsf{Logic}\ {:}{\wedge}{:}\ \mathsf{Logic} && \text{-- conjunction (and)} \\
\mid {} & \mathsf{Logic}\ {:}{\vee}{:}\ \mathsf{Logic} && \text{-- disjunction (or)} \\
\mid {} & \mathit{Not}\ \mathsf{Logic} && \text{-- negation (not)} \\
\mid {} & \mathit{Var}\ \mathsf{String} && \text{-- variables} \\
\mid {} & T && \text{-- true} \\
\mid {} & F && \text{-- false}
\end{aligned}
$$

# Exercise assistants: Logic III

$$showLogic :: \mathsf{Logic} \rightarrow \mathsf{String}$$
$$showLogic = \ldots$$

# Exercise assistants: Logic III

$$showLogic :: \mathsf{Logic} \rightarrow \mathsf{String}$$
$$showLogic = \ldots$$

$$parseLogic :: \mathsf{String} \rightarrow \mathsf{Logic}$$
$$parseLogic = \ldots$$

# Exercise assistants: Logic III

$$showLogic :: \mathsf{Logic} \rightarrow \mathsf{String}$$
$$showLogic = \ldots$$

$$parseLogic :: \mathsf{String} \rightarrow \mathsf{Logic}$$
$$parseLogic = \ldots$$

$$\mathbf{type}\ \mathsf{LogicAlgebra\ a} = \ldots$$
$$foldLogic :: \mathsf{LogicAlgebra\ a} \rightarrow \mathsf{Logic} \rightarrow \mathsf{a}$$
$$foldLogic = \ldots$$
$$evalLogic :: (\mathsf{String} \rightarrow \mathsf{Bool}) \rightarrow \mathsf{Logic} \rightarrow \mathsf{Bool}$$
$$evalLogic\ env\ l = foldLogic\ \ \ldots\ l$$

# Exercise assistants: Logic III

$$showLogic :: \text{Logic} \rightarrow \text{String}$$
$$showLogic = \ldots$$

$$parseLogic :: \text{String} \rightarrow \text{Logic}$$
$$parseLogic = \ldots$$

$$\textbf{type}\ \text{LogicAlgebra a} = \ldots$$

$$foldLogic :: \text{LogicAlgebra a} \rightarrow \text{Logic} \rightarrow \text{a}$$
$$foldLogic = \ldots$$

$$evalLogic :: (\text{String} \rightarrow \text{Bool}) \rightarrow \text{Logic} \rightarrow \text{Bool}$$
$$evalLogic\ env\ l = foldLogic\ \ldots\ l$$

$$\textbf{instance}\ Arbitrary\ \text{Logic}\ \textbf{where}$$
$$\quad arbitrary = \ldots$$

...

# Exercise assistants: Linear expressions I

Great! Your exercise assistant was a success and now you are asked to develop a tool to help students solving linear equations.

You need:

- A description of the linear expressions domain

# Exercise assistants: Linear expressions I

Great! Your exercise assistant was a success and now you are asked to develop a tool to help students solving linear equations.

You need:

- A description of the linear expressions domain
- Functionality on that domain:
  - Parsing and pretty-printing
  - Equality and top-level equality
  - Folding
  - Exercise generation
  - ...

# Exercise assistants: Linear expressions II

Let's get started, then:

```
data Expr = Con Rational      -- Constants
          | EVar String       -- Variables
          | Expr :+: Expr     -- Addition
          | Expr :−: Expr     -- Subtraction
          | Expr :×: Expr     -- Multiplication
          | Expr :/: Expr     -- Division
```

# Exercise assistants: Linear expressions III

$$showExpr :: \mathsf{Expr} \rightarrow \mathsf{String}$$
$$showExpr = \ldots$$

# Exercise assistants: Linear expressions III

$$showExpr :: \mathsf{Expr} \rightarrow \mathsf{String}$$
$$showExpr = \ldots$$

$$parseExpr :: \mathsf{String} \rightarrow \mathsf{Expr}$$
$$parseExpr = \ldots$$

# Exercise assistants: Linear expressions III

$$showExpr :: \mathsf{Expr} \rightarrow \mathsf{String}$$
$$showExpr = \ldots$$

$$parseExpr :: \mathsf{String} \rightarrow \mathsf{Expr}$$
$$parseExpr = \ldots$$

$$\textbf{type}\ ExprAlgebra\ \mathsf{a} = \ldots$$
$$foldExpr :: ExprAlgebra\ \mathsf{a} \rightarrow \mathsf{Expr} \rightarrow \mathsf{a}$$
$$foldExpr = \ldots$$
$$evalExpr :: (\mathsf{String} \rightarrow \mathsf{Rational}) \rightarrow \mathsf{Expr} \rightarrow \mathsf{Rational}$$
$$evalExpr\ env\ e = foldExpr\ \ \ldots\ e$$

# Exercise assistants: Linear expressions III

$$\begin{array}{l}
\mathit{showExpr} :: \mathsf{Expr} \rightarrow \mathsf{String} \\
\mathit{showExpr} = \ldots \\
\\
\mathit{parseExpr} :: \mathsf{String} \rightarrow \mathsf{Expr} \\
\mathit{parseExpr} = \ldots \\
\\
\textbf{type}\ \mathit{ExprAlgebra}\ \mathsf{a} = \ldots \\
\mathit{foldExpr} :: \mathit{ExprAlgebra}\ \mathsf{a} \rightarrow \mathsf{Expr} \rightarrow \mathsf{a} \\
\mathit{foldExpr} = \ldots \\
\mathit{evalExpr} :: (\mathsf{String} \rightarrow \mathsf{Rational}) \rightarrow \mathsf{Expr} \rightarrow \mathsf{Rational} \\
\mathit{evalExpr}\ \mathit{env}\ e = \mathit{foldExpr}\ \ \ldots\ e \\
\\
\textbf{instance}\ \mathit{Arbitrary}\ \mathsf{Expr}\ \textbf{where} \\
\quad \mathit{arbitrary} = \ldots
\end{array}$$

...

# Exercise assistants: Polynomials. . .

Oops. After all your tool should deal with polynomials too. You need to add exponentiation to your datatype:

# Exercise assistants: Polynomials. . .

Oops. After all your tool should deal with polynomials too. You need to add exponentiation to your datatype:

```
data Expr = Con Rational
          | EVar String
          | Expr :+: Expr
          | Expr :−: Expr
          | Expr :×: Expr
          | Expr :/: Expr
          | Expr :^: Expr   -- Exponentiation
```

# Exercise assistants: Polynomials. . .

Oops. After all your tool should deal with polynomials too. You need to add exponentiation to your datatype:

$$
\begin{aligned}
\textbf{data}\ \text{Expr} = &\ \mathit{Con}\ \text{Rational} \\
|\ &\ \mathit{EVar}\ \text{String} \\
|\ &\ \text{Expr} :+: \text{Expr} \\
|\ &\ \text{Expr} :-: \text{Expr} \\
|\ &\ \text{Expr} :\times: \text{Expr} \\
|\ &\ \text{Expr} :/: \text{Expr} \\
|\ &\ \text{Expr} :\hat{}: \text{Expr} \quad \text{-- Exponentiation}
\end{aligned}
$$

Of course, now you also need to change all your functions. . .

# Going generic

. . . is there no easier way to do this?. . .

# Going generic

. . . is there no easier way to do this?. . .

Yes! The answer is **Generic Programming**. With it you can:

- Write functions that work on any datatype
- Write common functionality once and for all
- Change your datatypes without changing your functions
- Avoid errors from code duplication
- . . .

# Ingredients for Generic Programming I

What is necessary for generic programming?

# Ingredients for Generic Programming I

What is necessary for generic programming?

The essential ingredient is a reflection mechanism. We have to be able to inspect values and their types at runtime.

# Ingredients for Generic Programming I

What is necessary for generic programming?

The essential ingredient is a reflection mechanism. We have to be able to inspect values and their types at runtime.

Additionally, we have to be able to represent many different values in a uniform way. If we can map all values into a small set of a datatypes, we can then define functions on this small set and they will work for every datatype.

# Ingredients for Generic Programming II

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

# Ingredients for Generic Programming II

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

We can represent them using the following data types:

$$
\begin{aligned}
&\textbf{data}\ \mathsf{a}\ {:+:}\ \mathsf{b} = L\ \mathsf{a} \mid R\ \mathsf{b} \\
&\textbf{data}\ \mathsf{a}\ {:\times:}\ \mathsf{b} = \mathsf{a}\ {:\times:}\ \mathsf{b} \\
&\textbf{data}\ \mathsf{Unit} \quad = \mathit{Unit}
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{infixr}\ 5\ {:+:} \\
&\textbf{infixr}\ 6\ {:\times:}
\end{aligned}
$$

# Structure Types

We can use these structure types to encode Haskell data types:

$$\textbf{data}\ \textsf{Tree} = \mathit{Leaf} \mid \mathit{Node}\ \textsf{Tree Int Tree}$$
$$\textbf{type}\ \textsf{RTree} = \textsf{Unit} :+: \textsf{Tree} :\times: \textsf{Int} :\times: \textsf{Tree}$$

$$\textbf{data}\ \textsf{List a} = \mathit{Nil} \mid \mathit{Cons}\ \textsf{a}\ (\textsf{List a})$$
$$\textbf{type}\ \textsf{RList a} = \textsf{Unit} :+: \textsf{a} :\times: \textsf{List a}$$

# Generic values

We encode the values in the same way:

$$tree :: \mathsf{Tree}$$
$$tree = Leaf$$
$$rtree :: \mathsf{RTree}$$
$$rtree = L\ Unit$$

$$list :: \mathsf{List\ Int}$$
$$list = Cons\ 2\ Nil$$
$$rlist :: \mathsf{RList\ Int}$$
$$rlist = R\ (2 :\times: Nil)$$

# Types and structure types are isomorphic

A type is isomorphic to its structural representation type. For example, for the list data type we have:

$$
\begin{aligned}
&\mathit{from}_{\mathsf{List}} :: \mathsf{List\ a} \rightarrow \mathsf{RList\ a}\\
&\mathit{from}_{\mathsf{List}}\ \mathit{Nil} = \mathit{L\ Unit}\\
&\mathit{from}_{\mathsf{List}}\ (\mathit{Cons\ a\ as}) = \mathit{R}\ (\mathit{a} :\times: \mathit{as})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{to}_{\mathsf{List}} :: \mathsf{RList\ a} \rightarrow \mathsf{List\ a}\\
&\mathit{to}_{\mathsf{List}}\ (\mathit{L\ Unit}) = \mathit{Nil}\\
&\mathit{to}_{\mathsf{List}}\ (\mathit{R}\ (\mathit{a} :\times: \mathit{as})) = \mathit{Cons\ a\ as}
\end{aligned}
$$

All the necessary infrastructure (RList, $\mathit{from}_{\mathsf{List}}$ and $\mathit{to}_{\mathsf{List}}$) can be generated automatically.

# Generic functions

A generic function can now be defined by induction on the structure of types, by writing cases for binary sums, binary products, nullary products, and primitives.

We use a GADT to unify the representation types into a single Rep:

$$
\begin{aligned}
&\textbf{data}\ \mathsf{Rep\ t}\ \textbf{where}\\
&\quad \mathit{RSum} :: \mathsf{Rep\ a} \rightarrow \mathsf{Rep\ b} \rightarrow \mathsf{Rep\ (a :\!+\!: b)}\\
&\quad \mathit{RProd} :: \mathsf{Rep\ a} \rightarrow \mathsf{Rep\ b} \rightarrow \mathsf{Rep\ (a :\!\times\!: b)}\\
&\quad \mathit{RUnit} :: \mathsf{Rep\ Unit}\\
&\quad \mathit{RInt} :: \mathsf{Rep\ Int}\\
&\quad \mathit{RChar} :: \mathsf{Rep\ Char}
\end{aligned}
$$

# Generic equality I

Now we can define, say, generic equality:

$$
\begin{array}{lllll}
eq :: \mathsf{Rep\ a} \to \mathsf{a} \to \mathsf{a} \to \mathsf{Bool} \\
eq\ (RInt\ \ ) & i & j & = & eq_{Int}\ i\ j \\
eq\ (RChar) & c & d & = & eq_{Char}\ c\ d \\
eq\ (RUnit) & Unit & Unit & = & True \\
eq\ (RSum\ r_a\ r_b) & (L\ a_1) & (L\ a_2) & = & eq\ r_a\ a_1\ a_2 \\
eq\ (RSum\ r_a\ r_b) & (R\ b_1) & (R\ b_2) & = & eq\ r_b\ b_1\ b_2 \\
eq\ (RSum\ r_a\ r_b) & \_ & \_ & = & False \\
eq\ (RProd\ r_a\ r_b) & (a_1 :\times: b_1) & (a_2 :\times: b_2) & = & \phantom{\wedge}\ eq\ r_a\ a_1\ a_2 \\
 & & & & \wedge\ eq\ r_b\ b_1\ b_2
\end{array}
$$

# Generic equality II

But we are still lacking a case for arbitrary datatypes. When two types are isomorphic, the corresponding isomorphisms can be stored as a pair of functions converting back and forth—an embedding-projection pair:

$$\mathbf{data}\ \mathsf{EP}\ \mathsf{d}\ \mathsf{r} = \mathit{EP}\ \{\mathit{from} :: (\mathsf{d} \to \mathsf{r}), \mathit{to} :: (\mathsf{r} \to \mathsf{d})\}$$

# Generic equality II

But we are still lacking a case for arbitrary datatypes. When two types are isomorphic, the corresponding isomorphisms can be stored as a pair of functions converting back and forth—an embedding-projection pair:

$$\textbf{data}\ \mathsf{EP\ d\ r} = \mathit{EP}\ \{\mathit{from} :: (\mathsf{d} \rightarrow \mathsf{r}), \mathit{to} :: (\mathsf{r} \rightarrow \mathsf{d})\}$$

We extend our representation type with a case for arbitrary types:

$$\begin{aligned}&\textbf{data}\ \mathsf{Rep\ t}\ \textbf{where}\\ &\quad \ldots\\ &\quad \mathit{RType} :: \mathsf{EP\ d\ r} \rightarrow \mathsf{Rep\ r} \rightarrow \mathsf{Rep\ d}\end{aligned}$$


Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

# Generic equality III

And add this case to the generic equality function:

$$eq :: \mathsf{Rep\ a} \rightarrow \mathsf{a} \rightarrow \mathsf{a} \rightarrow \mathsf{Bool}$$
$$\ldots$$
$$eq\ (RType\ ep\ r_a)\ t1\ t2 = eq\ r_a\ (from\ ep\ t1)\ (from\ ep\ t2)$$

# Generic equality III

And add this case to the generic equality function:

$$eq :: \mathsf{Rep\ a} \rightarrow \mathsf{a} \rightarrow \mathsf{a} \rightarrow \mathsf{Bool}$$
$$\ldots$$
$$eq\ (RType\ ep\ r_a)\ t1\ t2 = eq\ r_a\ (from\ ep\ t1)\ (from\ ep\ t2)$$

As an example, for lists we have:

$$rList :: \mathsf{Rep\ a} \rightarrow \mathsf{Rep\ (List\ a)}$$
$$rList\ r_a = RType\ (EP\ from_{\mathsf{List}}\ to_{\mathsf{List}})\ (RSum\ RUnit\ (RProd\ r_a\ (rList\ r_a)))$$

# Approaches to Generic Programming in Haskell

The basic principle here described can be explored in several different ways. We have seen a variant of Lightweight Implementation of Generics and Dynamics (LIGD). There are several other libraries for generic programming:

- Scrap Your Boilerplate (SYB)
- Uniplate
- Generics for the Masses (EMGM)
- Regular
- MultiRec
- . . . and at least 7 others

These libraries vary in expressiveness, ease of use and understanding, and underlying mechanisms used.

# Conclusions I

- Generic programming provides a way of reducing "boilerplate" code
- Functions are defined on the structure of datatypes and therefore work for every datatype
- If a datatype changes, the generic functions do not need to be adapted

A lot of work has been done in generic programming, and many functions are already available "for free", such as generation of test data, (basic) parsing and pretty-printing, rewriting, etc.

# Conclusions II

Current work at Utrecht University focuses on:

- Development of a powerful, easy to use and expressive generic programming library
- Applying generic programming to a large, showcase application
- Comparing performance of different approaches and investigating techniques for optimization of generic programs