Universiteit Utrecht

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

Generic Programming: what, why and how

José Pedro Magalhães

USCS 2009, 28/08/2009

What kind of generic?

In many languages, the function below is generic:

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

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:

```
data Logic = Logic :→: Logic   -- implication
           | Logic :↔: Logic   -- equivalence
           | Logic :∧: Logic   -- conjunction (and)
           | Logic :∨: Logic   -- disjunction (or)
           | Not Logic         -- negation (not)
           | Var String        -- variables
           | T                 -- true
           | F                 -- false
```

Exercise assistants: Logic III

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

Exercise assistants: Logic III

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

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

Exercise assistants: Logic III

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

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

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

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

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

Exercise assistants: Logic III

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

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

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

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

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

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

$$\ldots$$

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

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

Exercise assistants: Linear expressions III

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

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

Exercise assistants: Linear expressions III

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

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

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

Exercise assistants: Linear expressions III

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

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

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

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

$$\ldots$$

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:

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

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:

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

Structure Types

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

$$\textbf{data}\ \text{Tree} \quad = \mathit{Leaf} \mid \mathit{Node}\ \text{Tree Int Tree}$$
$$\textbf{type}\ \text{Tree}^{\circ} = \text{Sum Unit (Prod Tree (Prod Int Tree))}$$

$$\textbf{data}\ \text{List}\ \ \text{a} = \mathit{Nil} \mid \mathit{Cons}\ \text{a (List a)}$$
$$\textbf{type}\ \text{List}^{\circ}\ \text{a} = \text{Sum Unit (Prod a (List a))}$$

Generic values

We encode the values in the same way:

$$
\begin{aligned}
&tree :: \mathsf{Tree} \\
&tree = Leaf \\
&tree^{\circ} :: \mathsf{Tree}^{\circ} \\
&tree^{\circ} = Inl\ Unit
\end{aligned}
$$

$$
\begin{aligned}
&list :: \mathsf{List\ Int} \\
&list = Cons\ 2\ Nil \\
&list^{\circ} :: \mathsf{List}^{\circ}\ \mathsf{Int} \\
&list^{\circ} = Inr\ (2 :\times: Nil)
\end{aligned}
$$

Types and structure types are isomorphic

A type t and its structural representation type t° are isomorphic. For example, for the list data type we have:

$$
\begin{aligned}
&\mathit{from}_{\mathsf{List}} :: \mathsf{List\ a} \rightarrow \mathsf{List}^\circ\ \mathsf{a} \\
&\mathit{from}_{\mathsf{List}}\ \mathit{Nil} &&= \mathit{Inl}\ \mathit{Unit} \\
&\mathit{from}_{\mathsf{List}}\ (\mathit{Cons}\ a\ as) &&= \mathit{Inr}\ (a :\times: as) \\
&\mathit{to}_{\mathsf{List}} :: \mathsf{List}^\circ\ \mathsf{a} \rightarrow \mathsf{List\ a} \\
&\mathit{to}_{\mathsf{List}}\ (\mathit{Inl}\ \mathit{Unit}) &&= \mathit{Nil} \\
&\mathit{to}_{\mathsf{List}}\ (\mathit{Inr}\ (a :\times: as)) &&= \mathit{Cons}\ a\ as
\end{aligned}
$$

Types and structure types are isomorphic

A type t and its structural representation type t° are isomorphic. For example, for the list data type we have:

$$
\begin{aligned}
&\mathit{from}_{\mathsf{List}} :: \mathsf{List\ a} \rightarrow \mathsf{List}^\circ\ \mathsf{a} \\
&\mathit{from}_{\mathsf{List}}\ \mathit{Nil} &&= \mathit{Inl}\ \mathit{Unit} \\
&\mathit{from}_{\mathsf{List}}\ (\mathit{Cons}\ a\ as) &&= \mathit{Inr}\ (a :\times: as) \\
&\mathit{to}_{\mathsf{List}} :: \mathsf{List}^\circ\ \mathsf{a} \rightarrow \mathsf{List\ a} \\
&\mathit{to}_{\mathsf{List}}\ (\mathit{Inl}\ \mathit{Unit}) &&= \mathit{Nil} \\
&\mathit{to}_{\mathsf{List}}\ (\mathit{Inr}\ (a :\times: as)) &&= \mathit{Cons}\ a\ as
\end{aligned}
$$

When two types are isomorphic, the corresponding isomorphisms can be stored as a pair of functions converting back and forth—an embedding-projection pair:

$$\mathbf{data}\ \mathsf{EP\ a\ b} = \mathit{EP}\ \{\mathit{from} :: (\mathsf{a} \rightarrow \mathsf{b}), \mathit{to} :: (\mathsf{b} \rightarrow \mathsf{a})\}$$

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 can converts values to and from their generic representations as necessary. All the necessary infrastructure (like List°, $\mathit{from}_{\mathsf{List}}$ and $\mathit{to}_{\mathsf{List}}$) can even be generated automatically.

Approaches to Generic Programming in Haskell

The basic principle here described can be explored in several different ways. In Haskell there are several libraries for generic programming:

- Scrap Your Boilerplate
- Uniplate
- Generics for the Masses
- MultiRec
- . . . and at least 8 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