Universiteit Utrecht

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

Why Generic Programming Matters

José Pedro Magalhães Johan Jeuring

Dept. of Information and Computing Sciences, Utrecht University
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
Web pages: `http://www.cs.uu.nl/wiki/Center`

IPA Herfstdagen 2008, 27/11/2008

What is generic programming about? I

Software usually works with structured information:

- Web browsers (HTML documents)
- Text formatters (LaTeX, MS Word)
- Spreadsheets (Excel)

This structure can be represented by a datatype or a DTD (Document Type Definition) or a Schema.

If such a type changes, all programs that work on that type have to be changed too, although often the central problem does not change.

What is generic programming about? II

Some software tools do the same thing, modulo the structure of the data. Examples are webshops, compressors, exercise-assistants, ...

Generic programming deals with the methods with which problems can be formulated and solved for arbitrary datatypes or DTDs. The result is a generic program.

Changing Software

- Publisher example:
  - TEI: Text Encoding Initiative. Version 1: 1990, Version 2: 1993, Version 3: 1994, 1999, Version 4: 2002, Version 5: 2004, ...
- *Changing* software is difficult, and has led to the *Software Evolution* research field.
- Current approaches to software evolution focus on analyzing, visualizing and refactoring existing software.
- Theories about constructing software that automatically adapts to a changing environment are scarce and challenging.

Further applications of generic programming

Further applications of generic programming are found in:

- Haskell's deriving construct: equality, read/show, ...
- Compiler tools: debuggers, garbage collectors, tracers.
- Software testing tools.
- XML tools: editors, compressors, database tools.
- Typed term processing: rewriting, unification, pattern matching.
- Exercise assistants

Almost all of these fields have in common that structure drives the application. The programs are 'syntax-directed'.

Most software contains one or more components that can be grabbed off the 'generic-programming shelf'.

Generic programming in action

I will now discuss how we can use type-indexed functions to obtain generic functions. In order to do so, I will introduce:

- Structure types.
- Embedding-projection pairs.

I will demonstrate these concepts by implementing the generic function *encode* in Generic Haskell.

Generic Haskell is an extension of Haskell that makes generic programming more convenient.

encode on trees

$$\begin{aligned}
&\textbf{data}\ \mathsf{Tree} = \mathit{Leaf} \mid \mathit{Node}\ \mathsf{Tree}\ \mathsf{Int}\ \mathsf{Tree}\\
&\textbf{data}\ \mathsf{Bit}\ \ = O \mid I
\end{aligned}$$

$$\begin{aligned}
&\mathit{encodeTree} :: \mathsf{Tree} \to [\mathsf{Bit}]\\
&\mathit{encodeTree}\ \mathit{Leaf} = [O]\\
&\mathit{encodeTree}\ (\mathit{Node}\ t_1\ i\ t_2) = [I] +\!\!+ \mathit{encodeTree}\ t_1\\
&\qquad\qquad +\!\!+ \mathit{encodeInt}\ i\\
&\qquad\qquad +\!\!+ \mathit{encodeTree}\ t_2
\end{aligned}$$

$$\begin{aligned}
&\mathit{encodeInt} :: \mathsf{Int} \to [\mathsf{Bit}]\\
&\mathit{encodeInt}\ i = \ldots
\end{aligned}$$

Generic *encode*

How do we define *encode* in general?

We need to know how to handle:

- Different alternatives: disjoint sums.
- Arguments of a constructor: products.
- Constructors and field labels.
- Primitive types: Int, Char, ...
- Function space constructors.
- And maybe some other base types.

We need to translate every data type to the above set of constructs and apply the appropriate code in the right place—or get the compiler to do it.

Types for representing structure

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

Generic Haskell provides the following data types:

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

Structure Types

We can use these types to encode Haskell data types by means of structure types.

$$\begin{aligned}
&\textbf{data}\ \text{Tree} &&= \mathit{Leaf} \mid \mathit{Node}\ \text{Tree Int Tree}\\
&\textbf{type}\ \mathit{Str}\ (\text{Tree}) &&= \text{Sum Unit (Prod Tree (Prod Int Tree))}
\end{aligned}$$

$$\begin{aligned}
&\textbf{data}\ \text{List a} &&= \mathit{Nil} \mid \mathit{Cons}\ \text{a (List a)}\\
&\textbf{type}\ \mathit{Str}\ (\text{List})\ \text{a} &&= \text{Sum Unit (Prod a (List a))}
\end{aligned}$$

Embedding projection pairs

If two types are isomorphic, the corresponding isomorphisms can be stored as a pair of functions converting back and forth:

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

A value of this type is called an embedding-projection pair.

Types and structure types are isomorphic

A type t and its structural representation type $Str\ (\mathsf{t})$ are isomorphic. For example, for the list data type we have

$$conv_{\mathsf{List}} = EP\ from_{\mathsf{List}}\ to_{\mathsf{List}}$$

$$\begin{aligned}
&from_{\mathsf{List}} :: \mathsf{List\ a} \rightarrow Str\ (\mathsf{List})\ \mathsf{a} \\
&from_{\mathsf{List}}\ Nil &&= Inl\ Unit \\
&from_{\mathsf{List}}\ (Cons\ a\ as) &&= Inr\ (a \times as) \\
&to_{\mathsf{List}} &&:: Str\ (\mathsf{List})\ \mathsf{a} \rightarrow \mathsf{List\ a} \\
&to_{\mathsf{List}}\ (Inl\ Unit) &&= Nil \\
&to_{\mathsf{List}}\ (Inr\ (a \times as)) &&= Cons\ a\ as
\end{aligned}$$

Generic functions

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

The definition of generic *encode* is straightforward.

$$
\begin{array}{lll}
encode\ \{|\mathsf{a} :: *|\} :: (encode\ \{|\mathsf{a}|\}) \Rightarrow \mathsf{a} \rightarrow [\mathsf{Bit}] & & \\
encode\ \{|\mathsf{Int}|\} & & = encodeInt \\
encode\ \{|\mathsf{Char}|\} & & = encodeChar \\
encode\ \{|\mathsf{Unit}|\} & Unit & = [\,] \\
encode\ \{|\mathsf{Prod}\ \alpha\ \beta|\} & (x_1 \times x_2) & = encode\ \{|\alpha|\}\ x_1 +\!\!+ \\
 & & \quad encode\ \{|\beta|\}\ x_2 \\
encode\ \{|\mathsf{Sum}\ \alpha\ \beta|\} & (Inl\ x) & = O : encode\ \{|\alpha|\}\ x \\
encode\ \{|\mathsf{Sum}\ \alpha\ \beta|\} & (Inr\ x) & = I\ \ : encode\ \{|\beta|\}\ x
\end{array}
$$

Conclusions I

- Generic programming provides a way of increasing code resilience to changes
- 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 (and in particular my research) focuses on:

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