Generic programming — Or: write everything once

RALF HINZE

Institut für Informatik, Lehrstuhl Softwaretechnik, Universität Freiburg Georges-Köhler-Allee, Gebäude 079, 79110 Freiburg i. Br.

Email: ralf@informatik.uni-bonn.de
Homepage: http://www.informatik.uni-bonn.de/~ralf

4. Juli 2005

(Pick up the slides at .../~ralf/talks.html#T42.)
Motto:

Generic programming is about making programs more adaptable by making them more general.

Generic programming languages allow a wider range of entities as parameters than is available in more traditional languages: generic programs possibly abstract over

- other programs,
- types or type constructors,
- modules,
- classes,
- . . .
In this talk, we look at a particularly elegant instantiation of the idea of generic programming: **data-generic programming**.

**A data-generic program is a collection of functions and types that are defined by induction on the structure of types.**

**Benefits**: a data-generic program

- automatically adapts to changes in the representation of data,
  - partial answer to the problem of **software evolution**,  
- is usually simpler and more concise than a specific instance.
The concept of data-generic programming trades under a variety of names:

- structural polymorphism,
- type parametric programming,
- shape polymorphism,
- intensional polymorphism,
- polytypic programming.

Related lines of research:

- Standard Template Library (STL): parametric (or bounded) polymorphism,
- meta-programming: programs that manipulate other programs,
- reflection: ability of a program to examine and modify its structure.
Overview

A brief look at Haskell:

- Data types

A generic programming extension for Haskell:

- Generic functions on types
- Generic functions on type constructors
- Generic types
- Projects
Data types

In Haskell, a new type is introduced via a `data` declaration.

Examples:

```haskell
data Color = Red | Green | Blue
```

<Color is a type, Red, Green and Blue are data constructors.>

```haskell
data Maybe α = Nothing | Just α
```

```haskell
data Tree α = Empty | Node (Tree α) α (Tree α)
```

<α is a type parameter, Maybe and Tree are type constructors (functions on types).>
Data types — example: abstract syntax trees

```haskell
data Expr = Var Var -- variable
      | Nil -- nil
      | Num Integer -- numeral
      | String String -- string literal
      | Call Ident [Expr] -- function call
      | Un UnOp Expr -- unary operator
      | Bin Expr BinOp Expr -- binary operator
      | Record [(Ident, Expr)] TyIdent -- record creation
      | Array TyIdent Expr Expr -- array creation
      | Block [Expr] -- compound statement
      | Assign Var Expr -- assignment
      | IfThen Expr Expr -- one-sided alternative
      | IfElse Expr Expr Expr -- two-sided alternative
      | While Expr Expr -- while loop
      | For Ident Expr Expr Expr -- for loop
      | Let [Decl] Expr -- local definitions
      | Break -- loop exit
```
Algebraic data types are surprisingly expressive: the following definition captures the structural invariants of 2-3 trees: all leaves occur at the same level.

```haskell
data Node α = Node2 α α | Node3 α α α
data Tree23 α = Zero α | Succ (Tree23 (Node α))
```

Tree23 is a so-called nested data type.

OO: algebraic data types are closely related to the Composite pattern.
The structure of data types

What is the structure of algebraic data types?

Data types are built from primitive types (Integer, Char, etc) and three elementary types: the one-element type, binary sums, and binary products (below expressed as data types),

\[
\begin{align*}
\text{data } 1 & = () \\
\text{data } \alpha \times \beta & = (\alpha, \beta) \\
\text{data } \alpha + \beta & = \text{Inl } \alpha \mid \text{Inr } \beta
\end{align*}
\]

using type abstraction, type application, and type recursion.
Overview

A brief look at Haskell:

✔ Data types

A generic programming extension for Haskell:

✖ Generic functions on types
✖ Generic functions on type constructors
✖ Generic types
✖ Projects
A generic function is defined by induction on the structure of types.

Example: lexicographic ordering of values.

```haskell
data Ordering = LT | EQ | GT

compare :: (τ, τ) → Ordering
```

(compare is not a parametrically polymorphic function: a polymorphic function happens to be insensitive to what type the values in some structure are; the action of a generic function depends on the type argument.)
Generic functions — ordering

Implementing *compare* so that it works for arbitrary data types seems like a hard nut to crack. The good news is that it suffices to define *compare* for the three elementary types (the one-element type, binary sums, and binary products).

\[
\begin{align*}
\text{compare}(\alpha) &: (\alpha, \alpha) \rightarrow \text{Ordering} \\
\text{compare}(1) &: (() ,()) = \text{EQ} \\
\text{compare}(\alpha + \beta) &: (\text{Inl } a_1, \text{Inl } a_2) = \text{compare}(\alpha)(a_1, a_2) \\
\text{compare}(\alpha + \beta) &: (\text{Inl } a_1, \text{Inr } b_2) = \text{LT} \\
\text{compare}(\alpha + \beta) &: (\text{Inr } b_1, \text{Inl } a_2) = \text{GT} \\
\text{compare}(\alpha + \beta) &: (\text{Inr } b_1, \text{Inr } b_2) = \text{compare}(\beta)(b_1, b_2) \\
\text{compare}(\alpha \times \beta) &: ((a_1, b_1), (a_2, b_2)) = \text{case compare}(\alpha)(a_1, a_2) \text{ of} \\
& \quad \text{LT} \rightarrow \text{LT} \\
& \quad \text{EQ} \rightarrow \text{compare}(\beta)(b_1, b_2) \\
& \quad \text{GT} \rightarrow \text{GT}
\end{align*}
\]

☞ For emphasis, the type argument is enclosed in angle brackets.
Generic functions — examples

More examples:

- equality: deep equality,
- pretty printing: showing a value in a human-readable format,
- parsing: reading a value from a human-readable format,
- visualisation: converting a tree to a graphical representation,
- serialising (marshalling, pickling): conversion to an external data representation that can be transmitted across a network,
- data compression: conversion to a format that takes less space,
- generic traversals,
- ...
Overview

A brief look at Haskell:

✔ Data types

A generic programming extension for Haskell:

✔ Generic functions on types
✖ Generic functions on type constructors
✖ Generic types
✖ Projects
Generic functions on type constructors

The function *compare* abstracts over a type: it generalises functions of type

\[(\text{Color, Color}) \rightarrow \text{Ordering},\]
\[(\text{List Char, List Char}) \rightarrow \text{Ordering}\]
\[(\text{Tree Integer, Tree Integer}) \rightarrow \text{Ordering}\]
\[(\text{List (Tree Integer), List (Tree Integer)}) \rightarrow \text{Ordering}\]

... to a single generic function of type

\[\text{compare}\langle\alpha\rangle :: (\alpha, \alpha) \rightarrow \text{Ordering}\]
A generic function may also abstract over a type constructor.

For instance, the function that counts the number of elements in a container generalises functions of type

\[
\forall \alpha . \text{List } \alpha \rightarrow \text{Integer} \\
\forall \alpha . \text{Tree } \alpha \rightarrow \text{Integer} \\
\forall \alpha . \text{List} \ (\text{Tree } \alpha) \rightarrow \text{Integer} \\
\forall \alpha . \text{Tree23 } \alpha \rightarrow \text{Integer} \\
\ldots
\]

to a single generic function of type

\[
\text{size} \langle \varphi \rangle :: \forall \alpha . \varphi \ alpha \rightarrow \text{Integer}
\]
The definition of $size$ proceeds in two steps:

\[
\begin{align*}
\text{count}(\alpha) &:: \alpha \rightarrow \text{Integer} \\
\text{count}(1) &() = 0 \\
\text{count}(\alpha + \beta)(\text{Inl } a) &= \text{count}(\alpha) a \\
\text{count}(\alpha + \beta)(\text{Inr } b) &= \text{count}(\beta) b \\
\text{count}(\alpha \times \beta)(a, b) &= \text{count}(\alpha) a + \text{count}(\beta) b
\end{align*}
\]

\[
\begin{align*}
\text{size}(\varphi) &:: \forall \alpha . \varphi \alpha \rightarrow \text{Integer} \\
\text{size}(\varphi) &= \text{count}(\varphi \alpha) \textbf{ where } \text{count}(\alpha) a = 1
\end{align*}
\]
Generic functions — examples

More examples:

► many list processing functions can be generalised to arbitrary data types: `sum`, `product`, `and`, `or`, `forall`, `exists`, ...

► container conversion,

► reduction with a monoid: a reduction is a function that collapses a structure of type $\varphi \alpha$ into a single value of type $\alpha$,

► mapping function: a mapping function takes a function and applies it to each element of a given container, leaving its structure intact,

► mapping functions with effects,

► ...

☞ Reductions and mapping functions are related to the Visitor and Iterator pattern.
Overview

A brief look at Haskell:

✔  Data types

A generic programming extension for Haskell:

✔  Generic functions on types
✔  Generic functions on type constructors
✖  Generic types
✖  Projects
Generic types

A generic data type is a type that is defined by induction on the structure of an argument data type.

Example: Digital search trees, also known as tries, employ the structure of search keys to organise information.

\{\text{ear, earl, east, easy, eye}\} \implies \begin{array}{cc}
\text{\text{earl, east, easy, eye}} & \\
\text{\text{ear}} & \\
\text{\text{e}} & \\
\end{array}

A trie can be seen as a composition of finite maps.
Digital search trees are based on the laws of exponentials.

\[
\begin{align*}
1 \to_{\text{fin}} \nu & \equiv \nu \\
(\kappa_1 + \kappa_2) \to_{\text{fin}} \nu & \equiv (\kappa_1 \to_{\text{fin}} \nu) \times (\kappa_2 \to_{\text{fin}} \nu) \\
(\kappa_1 \times \kappa_2) \to_{\text{fin}} \nu & \equiv \kappa_1 \to_{\text{fin}} (\kappa_2 \to_{\text{fin}} \nu)
\end{align*}
\]

Using the laws of exponentials we can define a generic type of finite maps: \(\text{Map}\langle K \rangle V\) represents \(K \to_{\text{fin}} V\).

\[
\begin{align*}
data \text{Map}\langle 1 \rangle \nu & = \text{Maybe } \nu \\
data \text{Map}\langle \alpha + \beta \rangle \nu & = \text{Map}\langle \alpha \rangle \nu \times \text{Map}\langle \beta \rangle \nu \\
data \text{Map}\langle \alpha \times \beta \rangle \nu & = \text{Map}\langle \alpha \rangle (\text{Map}\langle \beta \rangle \nu)
\end{align*}
\]

The two type arguments of \(\text{Map}\) play different rôles: \(\text{Map}\langle K \rangle V\) is defined by induction on the structure of \(K\), but is parametric in \(V\).
Generic types — tries: lookup

A generic look-up function:

\[
\text{lookup}\langle \kappa \rangle :: \forall \nu . \kappa \to \text{Map}\langle \kappa \rangle \nu \to \text{Maybe} \nu
\]

\[
\text{lookup}\langle 1 \rangle \quad () \quad t = t
\]

\[
\text{lookup}\langle \alpha + \beta \rangle \quad (\text{Inl} \ a) \quad (ta, tb) = \text{lookup}\langle \alpha \rangle \ a \ ta
\]

\[
\text{lookup}\langle \alpha + \beta \rangle \quad (\text{Inr} \ b) \quad (ta, tb) = \text{lookup}\langle \beta \rangle \ b \ tb
\]

\[
\text{lookup}\langle \alpha \times \beta \rangle \quad (a, b) \quad ta = \text{case} \ \text{lookup}\langle \alpha \rangle \ a \ ta \ \text{of}
\]

\[
\text{Nothing} \to \text{Nothing}
\]

\[
\text{Just} \ tb \to \text{lookup}\langle \beta \rangle \ b \ tb
\]

Interesting instances: \(\text{Map}\langle \text{List Char} \rangle\) is the type of ‘conventional’ tries, \(\text{Map}\langle \text{List Bit} \rangle\) is the type of binary tries.
More examples:

- Memo functions.
- **Xcomprez**: compression of XML documents that are structured according to a Document Type Definition (DTD).
  - compression ratio 40%–50% better than XMill,
  - 300 lines of Generic Haskell versus 20,000 lines of C++ for XMill,
  - uses **HAXML** to translate DTDs into data types.
- Trees with a focus of interest (navigation trees, zipper, finger, pointer reversal): used in editors for structured documents, theorem provers.
- Labelled or decorated trees.
- Data parallel arrays: flattening transformation for nested data parallelism.
- ...
Overview

A brief look at Haskell:

✔ Data types

A generic programming extension for Haskell:

✔ Generic functions on types
✔ Generic functions on type constructors
✔ Generic types
✖ Projects
Projects


- Generic Haskell is an extension of Haskell that supports the construction of generic programs. The examples above are written in Generic Haskell.
- Generic Haskell is implemented as a preprocessor ($\approx 24,000$ LOC) that translates generic functions into Haskell.
- See www.generic-haskell.org.


- Integration of the extension into a standard Haskell compiler.
- Applications:
  - refactoring,
  - XML tools,
  - automatic testing,
  - data conversion.
- Generic specifications and proofs.
Overview

A brief look at Haskell:

- ✔ Data types

A generic programming extension for Haskell:

- ✔ Generic functions on types
- ✔ Generic functions on type constructors
- ✔ Generic types
- ✔ Projects
Conclusion

- Generic functions and types are defined by induction on the structure of types. The examples above are executable.

- A generic definition can be specialised to an arbitrary data type. It automatically adapts to changes in the representation of data.

- The generic definitions are statically typed; static typing guarantees that every instance will be well-typed.

- Generic programming, albeit more abstract, is often simpler and more concise than ordinary programming: we only have to provide instances for three simple, non-recursive data types.


Nested 2-3 trees explained

The data constructors \( \text{Zero} \) and \( \text{Succ} \) have types:

\[
\text{Zero} :: \forall \alpha . \alpha \rightarrow \text{Tree}_{23} \alpha \\
\text{Succ} :: \forall \alpha . \text{Tree}_{23} (\text{Node} \ \alpha) \rightarrow \text{Tree}_{23} \alpha
\]

Read the types as a term rewriting system.

Bottom-up construction of a 2-3 tree of height 2.

\[
\text{N}2 \ (\text{N}2 \ 1 \ 2) \ (\text{N}2 \ 2 \ 3) \ :: \ \text{Node} \ (\text{Node} \ \text{Int}) \\
\text{Zero} \ (\text{N}2 \ (\text{N}2 \ 1 \ 2) \ (\text{N}2 \ 2 \ 3)) \ :: \ \text{Tree}_{23} \ (\text{Node} \ (\text{Node} \ \text{Int})) \\
\text{Succ} \ (\text{Zero} \ (\text{N}2 \ (\text{N}2 \ 1 \ 2) \ (\text{N}2 \ 2 \ 3))) \ :: \ \text{Tree}_{23} \ (\text{Node} \ \text{Int}) \\
\text{Succ} \ (\text{Succ} \ (\text{Zero} \ (\text{N}2 \ (\text{N}2 \ 1 \ 2) \ (\text{N}2 \ 2 \ 3)))) \ :: \ \text{Tree}_{23} \ \text{Int}
\]

The type in the first line mirrors the structure of the tree on the type level.