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`.)

Introduction

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,
- ...

Introduction — continued

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.

Related work

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:

$$\textbf{data}\ \mathit{Color} = \mathit{Red} \mid \mathit{Green} \mid \mathit{Blue}$$

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

$$\textbf{data}\ \mathit{Maybe}\ \alpha = \mathit{Nothing} \mid \mathit{Just}\ \alpha$$

$$\textbf{data}\ \mathit{Tree}\ \alpha \quad = \mathit{Empty} \mid \mathit{Node}\ (\mathit{Tree}\ \alpha)\ \alpha\ (\mathit{Tree}\ \alpha)$$

☞ α is a type parameter, Maybe and Tree are type constructors (functions on types).

Data types — example: abstract syntax trees

```
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
```

Data types — example: nested types

Algebraic data types are surprisingly expressive: the following definition captures the structural invariants of 2-3 trees: all leaves occur at the same level.

$$\begin{aligned}
&\mathbf{data}\ \mathit{Node}\ \alpha &&= \mathit{Node2}\ \alpha\ \alpha \mid \mathit{Node3}\ \alpha\ \alpha\ \alpha \\
&\mathbf{data}\ \mathit{Tree23}\ \alpha &&= \mathit{Zero}\ \alpha \mid \mathit{Succ}\ (\mathit{Tree23}\ (\mathit{Node}\ \alpha))
\end{aligned}$$

☞ *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),

$$\begin{aligned}
&\textbf{data}\ 1 &&= () \\
&\textbf{data}\ \alpha \times \beta &&= (\alpha, \beta) \\
&\textbf{data}\ \alpha + \beta &&= \mathit{Inl}\ \alpha \mid \mathit{Inr}\ \beta
\end{aligned}$$

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

◀◀ ◀ ▶ ▶▶ ☐

Generic functions on types

A generic function is defined by induction on the structure of types.

Example: lexicographic ordering of values.

$$\textbf{data}\ \mathit{Ordering} = \mathit{LT} \mid \mathit{EQ} \mid \mathit{GT}$$

$$\mathit{compare} :: (\tau, \tau) \to \mathit{Ordering}$$

☞ *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).

$$
\begin{array}{llll}
\mathit{compare}\langle\alpha\rangle :: & (\alpha, \quad \alpha) & \rightarrow & \mathit{Ordering} \\
\mathit{compare}\langle 1\rangle & ((), \quad ()) & = & \mathit{EQ} \\
\mathit{compare}\langle\alpha+\beta\rangle & (\mathit{Inl}\ a_1, \ \mathit{Inl}\ a_2) & = & \mathit{compare}\langle\alpha\rangle\ (a_1, a_2) \\
\mathit{compare}\langle\alpha+\beta\rangle & (\mathit{Inl}\ a_1, \ \mathit{Inr}\ b_2) & = & \mathit{LT} \\
\mathit{compare}\langle\alpha+\beta\rangle & (\mathit{Inr}\ b_1, \ \mathit{Inl}\ a_2) & = & \mathit{GT} \\
\mathit{compare}\langle\alpha+\beta\rangle & (\mathit{Inr}\ b_1, \ \mathit{Inr}\ b_2) & = & \mathit{compare}\langle\beta\rangle\ (b_1, b_2) \\
\mathit{compare}\langle\alpha\times\beta\rangle & ((a_1, b_1), (a_2, b_2)) & = & \mathbf{case}\ \mathit{compare}\langle\alpha\rangle\ (a_1, a_2)\ \mathbf{of} \\
 & & & \quad \mathit{LT} \rightarrow \mathit{LT} \\
 & & & \quad \mathit{EQ} \rightarrow \mathit{compare}\langle\beta\rangle\ (b_1, b_2) \\
 & & & \quad \mathit{GT} \rightarrow \mathit{GT}
\end{array}
$$

☞ 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

$$
\begin{array}{l}
(Color, Color) \to Ordering, \\
(List\ Char, List\ Char) \to Ordering \\
(Tree\ Integer, Tree\ Integer) \to Ordering \\
(List\ (Tree\ Integer), List\ (Tree\ Integer)) \to Ordering \\
\ldots
\end{array}
$$

to a single generic function of type

$$compare\langle\alpha\rangle :: (\alpha, \alpha) \to Ordering$$

Generic functions — continued

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

$$
\begin{array}{l}
\forall \alpha \,.\, \mathit{List}\ \alpha \rightarrow \mathit{Integer} \\
\forall \alpha \,.\, \mathit{Tree}\ \alpha \rightarrow \mathit{Integer} \\
\forall \alpha \,.\, \mathit{List}\ (\mathit{Tree}\ \alpha) \rightarrow \mathit{Integer} \\
\forall \alpha \,.\, \mathit{Tree23}\ \alpha \rightarrow \mathit{Integer} \\
\ldots
\end{array}
$$

to a single generic function of type

$$\mathit{size}\langle \varphi \rangle :: \forall \alpha \,.\, \varphi\ \alpha \rightarrow \mathit{Integer}$$

Generic functions — size

The definition of size proceeds in two steps:

$$\begin{array}{llcl}
\mathit{count}\langle\alpha\rangle :: & \alpha & \to & \mathit{Integer} \\
\mathit{count}\langle 1\rangle & () & = & 0 \\
\mathit{count}\langle\alpha+\beta\rangle & (\mathit{Inl}\ a) & = & \mathit{count}\langle\alpha\rangle\ a \\
\mathit{count}\langle\alpha+\beta\rangle & (\mathit{Inr}\ b) & = & \mathit{count}\langle\beta\rangle\ b \\
\mathit{count}\langle\alpha\times\beta\rangle & (a, b) & = & \mathit{count}\langle\alpha\rangle\ a + \mathit{count}\langle\beta\rangle\ b
\end{array}$$

$$\mathit{size}\langle\varphi\rangle :: \forall\alpha\,.\,\varphi\ \alpha \to \mathit{Integer}$$

$$\mathit{size}\langle\varphi\rangle = \mathit{count}\langle\varphi\ \alpha\rangle\ \mathbf{where}\ \mathit{count}\langle\alpha\rangle\ a = 1$$

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 α,
- 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:

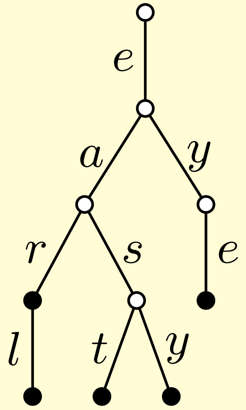
- ✔ Generic functions on types
- ✔ Generic functions on type constructors
- ✘ Generic types
- ✘ Projects

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.

$$\{ear,\ earl,\ east,\ easy,\ eye\} \Longrightarrow$$



A trie can be seen as a composition of finite maps.

Generic types — tries

Digital search trees are based on the laws of exponentials.

$$\begin{array}{lll}
1 & \to_{\mathrm{fin}} \nu \cong & \nu \\
(\kappa_1 + \kappa_2) & \to_{\mathrm{fin}} \nu \cong & (\kappa_1 \to_{\mathrm{fin}} \nu) \times (\kappa_2 \to_{\mathrm{fin}} \nu) \\
(\kappa_1 \times \kappa_2) & \to_{\mathrm{fin}} \nu \cong & \kappa_1 \to_{\mathrm{fin}} (\kappa_2 \to_{\mathrm{fin}} \nu)
\end{array}$$

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

$$\begin{array}{lll}
\mathbf{data}\ \mathit{Map}\langle 1\rangle & \nu = & \mathit{Maybe}\ \nu \\
\mathbf{data}\ \mathit{Map}\langle \alpha + \beta\rangle & \nu = & \mathit{Map}\langle\alpha\rangle\ \nu \times \mathit{Map}\langle\beta\rangle\ \nu \\
\mathbf{data}\ \mathit{Map}\langle \alpha \times \beta\rangle & \nu = & \mathit{Map}\langle\alpha\rangle\ (\mathit{Map}\langle\beta\rangle\ \nu)
\end{array}$$

☞ The two type arguments of Map play different rôles: $\mathit{Map}\langle K\rangle\ V$ is defined by induction on the structure of K, but is parametric in V.

Generic types — tries: lookup

A generic look-up function:

$$
\begin{array}{lllll}
\mathit{lookup}\langle\kappa\rangle :: \forall\nu\,.\,\kappa \to & & \mathit{Map}\langle\kappa\rangle\ \nu & \to & \mathit{Maybe}\ \nu \\
\mathit{lookup}\langle 1\rangle & () & t & = & t \\
\mathit{lookup}\langle\alpha+\beta\rangle & (\mathit{Inl}\ a) & (\mathit{ta},\mathit{tb}) & = & \mathit{lookup}\langle\alpha\rangle\ a\ \mathit{ta} \\
\mathit{lookup}\langle\alpha+\beta\rangle & (\mathit{Inr}\ b) & (\mathit{ta},\mathit{tb}) & = & \mathit{lookup}\langle\beta\rangle\ b\ \mathit{tb} \\
\mathit{lookup}\langle\alpha\times\beta\rangle & (a,b) & \mathit{ta} & = & \mathbf{case}\ \mathit{lookup}\langle\alpha\rangle\ a\ \mathit{ta}\ \mathbf{of} \\
 & & & & \quad \mathit{Nothing} \to \mathit{Nothing} \\
 & & & & \quad \mathit{Just}\ \mathit{tb} \ \ \to \mathit{lookup}\langle\beta\rangle\ b\ \mathit{tb}
\end{array}
$$

☞ Interesting instances: $\mathit{Map}\langle\mathit{List}\ \mathit{Char}\rangle$ is the type of ‘conventional’ tries, $\mathit{Map}\langle\mathit{List}\ \mathit{Bit}\rangle$ is the type of binary tries.

Generic data types — examples

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

Past: Generic Haskell (2000–2004, supported by: NWO).

- 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`.

Future: Eine generische funktionale Programmiersprache: Theorie, Sprachentwurf, Implementierung und Anwendung (Aug. 2005–Jul. 2007, supported by: DFG).

- 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.

Literature

- Ralf Hinze. A new approach to generic functional programming. In Thomas W. Reps, editor, *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts, January 19–21*, 2000, pp. 119–132.
- Ralf Hinze. Polytypic values possess polykinded types. *Science of Computer Programming*, MPC Special Issue, 42:129–159, 2002.
- Ralf Hinze and Johan Jeuring. *Generic Haskell: Practice and Theory*. In Roland Backhouse and Jeremy Gibbons, editors, *Generic Programming: Advanced Lectures*, pp.1–56. Lecture Notes in Computer Science 2793. Springer-Verlag, 2003.
- Ralf Hinze and Johan Jeuring. *Generic Haskell: Applications*. In Roland Backhouse and Jeremy Gibbons, editors, *Generic Programming: Advanced Lectures*, pp.57–97. Lecture Notes in Computer Science 2793. Springer-Verlag, 2003.
- Ralf Hinze, Johan Jeuring, and Andres Löh. Type-indexed data types. *Science of Computer Programmming*, MPC Special Issue, 51:117–151, 2004.

Nested 2-3 trees explained

The **data** constructors *Zero* and *Succ* have types:

$$\begin{aligned}
&\mathit{Zero} :: \forall \alpha \,.\, \alpha && \rightarrow \mathit{Tree23}\ \alpha \\[1em]
&\mathit{Succ} :: \forall \alpha \,.\, \mathit{Tree23}\ (\mathit{Node}\ \alpha) && \rightarrow \mathit{Tree23}\ \alpha
\end{aligned}$$

☞ Read the types as a term rewriting system.

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

$$\begin{aligned}
\mathit{N2}\ (\mathit{N2}\ 1\ 2)\ (\mathit{N2}\ 2\ 3) \quad &:: \mathit{Node}\ (\mathit{Node}\ \mathit{Int}) \\
\mathit{Zero}\ (\mathit{N2}\ (\mathit{N2}\ 1\ 2)\ (\mathit{N2}\ 2\ 3)) \quad &:: \mathit{Tree23}\ (\mathit{Node}\ (\mathit{Node}\ \mathit{Int})) \\
\mathit{Succ}\ (\mathit{Zero}\ (\mathit{N2}\ (\mathit{N2}\ 1\ 2)\ (\mathit{N2}\ 2\ 3))) \quad &:: \mathit{Tree23}\ (\mathit{Node}\ \mathit{Int}) \\
\mathit{Succ}\ (\mathit{Succ}\ (\mathit{Zero}\ (\mathit{N2}\ (\mathit{N2}\ 1\ 2)\ (\mathit{N2}\ 2\ 3)))) \quad &:: \mathit{Tree23}\ \mathit{Int}
\end{aligned}$$

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