Generic Haskell

Andres Löh
Universiteit Utrecht
andres@cs.uu.nl

24th November 2003

Overview

- Why Generic Haskell?
- Genericity and other types of polymorphism
- Writing and using generic functions in Generic Haskell
- Examples

Overview

- Why Generic Haskell?
- Genericity and other types of polymorphism
- Writing and using generic functions in Generic Haskell
- Examples

Haskell

- Haskell is a statically typed, pure functional language with lazy evaluation.

Haskell

- Haskell is a statically typed language.

Haskell

- Haskell is a statically typed language.
- Functions are defined by pattern matching.

Haskell

- Haskell is a statically typed language.
- Functions are defined by pattern matching.

$$
\begin{aligned}
&\mathit{factorial}\ \ 0 && = 1 \\
&\mathit{factorial}\ \ n && = n \cdot \mathit{factorial}\ (n-1)
\end{aligned}
$$

Haskell

- Haskell is a statically typed language.
- Functions are defined by pattern matching.

$$\begin{aligned} &\mathit{factorial}\ 0 &&= 1 \\ &\mathit{factorial}\ n &&= n \cdot \mathit{factorial}\ (n-1) \end{aligned}$$

- Every function has a type that usually can be inferred by the compiler.

$$\mathit{factorial} :: \mathsf{Int} \rightarrow \mathsf{Int}$$

Haskell

- Haskell is a statically typed language.
- Functions are defined by pattern matching.

$$
\begin{aligned}
&\mathit{factorial}\ 0 &&= 1\\
&\mathit{factorial}\ n &&= n \cdot \mathit{factorial}\ (n-1)
\end{aligned}
$$

- Every function has a type that usually can be inferred by the compiler.

$$\mathit{factorial} :: \mathsf{Int} \rightarrow \mathsf{Int}$$

- Functions with multiple arguments are written in curried style.

$$
\begin{aligned}
&\mathit{and} &&:: \mathsf{Bool} \rightarrow \mathsf{Bool} \rightarrow \mathsf{Bool}\\
&\mathit{and}\ \mathit{True}\ \mathit{True} &&= \mathit{True}\\
&\mathit{and}\ \_\ \ \_ &&= \mathit{False}
\end{aligned}
$$

User-defined datatypes

- Own datatypes can be defined in Haskell using the **data** construct:

```
data Nat = Zero | Succ Nat
```

User-defined datatypes

- Own datatypes can be defined in Haskell using the **data** construct:

data $\mathsf{Nat} = \mathit{Zero} \mid \mathit{Succ}\ \mathsf{Nat}$

$\mathit{Succ}\ (\mathit{Succ}\ (\mathit{Succ}\ \mathit{Zero}))$ represents the number 3

User-defined datatypes

- Own datatypes can be defined in Haskell using the **data** construct:

$\mathbf{data}\ \mathsf{Nat} = \mathit{Zero} \mid \mathit{Succ}\ \mathsf{Nat}$

$\mathit{Succ}\ (\mathit{Succ}\ (\mathit{Succ}\ \mathit{Zero}))$ represents the number 3

- Functions are often defined recursively over the structure of a datatype:

$$
\begin{array}{ll}
\mathit{plus} & :: \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat} \\
\mathit{plus}\ m\ \mathit{Zero} & = m \\
\mathit{plus}\ m\ (\mathit{Succ}\ n) & = \mathit{Succ}\ (\mathit{plus}\ m\ n)
\end{array}
$$

Why Generic Haskell?

Among other things, there are two desireable goals for programming languages:

- Abstraction
- Static guarantees

About abstraction

- Extract common patterns.
- Examples:
  - Loops
  - Modules
  - Functions
  - Classes
  - Higher-order functions
- Advantages:
  - Consistency
  - Testing/correctness
  - Reuse
  - Conciseness

About static guarantees

- Syntactical correctness
- Scoping rules/unbound identifiers
- Static typing
- Advantages:
  - Efficiency
  - Safety
  - Higher quality of resulting product
  - Testing

Static typing prevents abstraction

- It is impossible to analyze all interesting properties of a program at compile time (halting problem).
- A safe type system is necessarily conservative. It rejects programs that work, or prevents you to write programs you want to write.
- In dynamically typed languages, this problem does not occur.
- Generic Haskell can therefore be seen as an attempt to provide a stronger type system to allow more abstraction while maintaining safety.

Overview

- Why Generic Haskell?
- Genericity and other types of polymorphism.
- Writing and using generic functions in Generic Haskell
- Examples

Haskell datatypes

Haskell's **data** construct is extremely flexible. Here are a few example datatypes:

data TimeInfo $= AM \mid PM \mid H24$

Haskell datatypes

Haskell's **data** construct is extremely flexible. Here are a few example datatypes:

data TimeInfo $= AM \mid PM \mid H24$

data PackageDesc $= PD$ String Author Version Date

Haskell datatypes

Haskell's **data** construct is extremely flexible. Here are a few example datatypes:

data TimeInfo $= AM \mid PM \mid H24$

data PackageDesc $= PD$ String Author Version Date

data Package $= P$ PackageDesc $[$Package$]$

Haskell datatypes

Haskell's **data** construct is extremely flexible. Here are a few example datatypes:

data TimeInfo $= AM \mid PM \mid H24$

data PackageDesc $= PD$ String Author Version Date

data Package $= P$ PackageDesc [Package]

data Maybe α $= Nothing \mid Just\ \alpha$

Haskell datatypes

Haskell's **data** construct is extremely flexible. Here are a few example datatypes:

data TimeInfo $= AM \mid PM \mid H24$

data PackageDesc $= PD$ String Author Version Date

data Package $= P$ PackageDesc [Package]

data Maybe α $= Nothing \mid Just\ \alpha$

data Tree α $= Leaf\ \alpha \mid Node$ (Tree α) (Tree α)

Haskell datatypes

Haskell's **data** construct is extremely flexible. Here are a few example datatypes:

data TimeInfo $= AM \mid PM \mid H24$

data PackageDesc $= PD$ String Author Version Date

data Package $= P$ PackageDesc [Package]

data Maybe α $= Nothing \mid Just\ \alpha$

data Tree α $= Leaf\ \alpha \mid Node$ (Tree α) (Tree α)

data Perfect α $= ZeroP\ \alpha \mid SuccP$ (Perfect (α, α))

Haskell datatypes

Haskell's **data** construct is extremely flexible. Here are a few example datatypes:

data TimeInfo $= \mathit{AM} \mid \mathit{PM} \mid \mathit{H24}$

data PackageDesc $= \mathit{PD}$ String Author Version Date

data Package $= P$ PackageDesc [Package]

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

data Tree $\alpha = \mathit{Leaf}\ \alpha \mid \mathit{Node}\ (\mathsf{Tree}\ \alpha)\ (\mathsf{Tree}\ \alpha)$

data Perfect $\alpha = \mathit{ZeroP}\ \alpha \mid \mathit{SuccP}\ (\mathsf{Perfect}\ (\alpha, \alpha))$

data Compose $\varphi\ \psi\ \alpha = \mathit{Comp}\ (\varphi\ (\psi\ \alpha))$

Haskell datatypes

Haskell's **data** construct is extremely flexible. Here are a few example datatypes:

$$
\begin{aligned}
&\textbf{data}\ \mathsf{TimeInfo} &&= \mathit{AM} \mid \mathit{PM} \mid \mathit{H24} \\
&\textbf{data}\ \mathsf{PackageDesc} &&= \mathit{PD}\ \mathsf{String}\ \mathsf{Author}\ \mathsf{Version}\ \mathsf{Date} \\
&\textbf{data}\ \mathsf{Package} &&= \mathit{P}\ \mathsf{PackageDesc}\ [\mathsf{Package}] \\
&\textbf{data}\ \mathsf{Maybe}\ \alpha &&= \mathit{Nothing} \mid \mathit{Just}\ \alpha \\
&\textbf{data}\ \mathsf{Tree}\ \alpha &&= \mathit{Leaf}\ \alpha \mid \mathit{Node}\ (\mathsf{Tree}\ \alpha)\ (\mathsf{Tree}\ \alpha) \\
&\textbf{data}\ \mathsf{Perfect}\ \alpha &&= \mathit{ZeroP}\ \alpha \mid \mathit{SuccP}\ (\mathsf{Perfect}\ (\alpha, \alpha)) \\
&\textbf{data}\ \mathsf{Compose}\ \varphi\ \psi\ \alpha &&= \mathit{Comp}\ (\varphi\ (\psi\ \alpha))
\end{aligned}
$$

Nevertheless, datatypes have a common structure: parametrized over a number of **arguments**, several **constructors** mark multiple **alternatives**, each constructor has multiple **fields**, and there may be recursion.

Parametric polymorphism

Haskell allows to express functions that work uniformly on all datatypes.

$$
\begin{aligned}
&id && :: \forall\alpha.\alpha \to \alpha \\
&id\ x && = x
\end{aligned}
$$

Parametric polymorphism

Haskell allows to express functions that work uniformly on all datatypes.

$$
\begin{array}{ll}
\mathit{id} & :: \forall\alpha.\alpha \rightarrow \alpha \\
\mathit{id}\ x & = x
\end{array}
$$

$$
\begin{array}{ll}
\mathit{swap} & :: \forall\alpha\ \beta.(\alpha,\beta) \rightarrow (\beta,\alpha) \\
\mathit{swap}\ (x,y) & = (y,x)
\end{array}
$$

Parametric polymorphism

Haskell allows to express functions that work uniformly on all datatypes.

$$
\begin{aligned}
&\mathit{id} && :: \forall\alpha.\alpha \to \alpha \\
&\mathit{id}\ x && = x
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{swap} && :: \forall\alpha\ \beta.(\alpha,\beta) \to (\beta,\alpha) \\
&\mathit{swap}\ (x,y) && = (y,x)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{head} && :: \forall\alpha.[\alpha] \to \alpha \\
&\mathit{head}\ (x:xs) && = x
\end{aligned}
$$

We can take the *head* of a list of Packages, or swap a tuple of two Perfect trees.

What about equality?

- We know intuitively what it means for two Packages to be equal.
- We also know what it means for two Perfect trees, normal Trees, Maybes or TimeInfos to be equal.

What about equality?

- We know intuitively what it means for two Packages to be equal.
- We also know what it means for two Perfect trees, normal Trees, Maybes or TimeInfos to be equal.

Can you give a parametrically polymorphic definition for equality?

What about equality?

- We know intuitively what it means for two Packages to be equal.
- We also know what it means for two Perfect trees, normal Trees, Maybes or TimeInfos to be equal.

Can you give a parametrically polymorphic definition for equality?

$$
\begin{aligned}
&(\equiv) \quad :: \forall \alpha . \alpha \rightarrow \alpha \rightarrow \mathsf{Bool} \\
&x \equiv y = ???
\end{aligned}
$$

What about equality?

- We know intuitively what it means for two Packages to be equal.
- We also know what it means for two Perfect trees, normal Trees, Maybes or TimeInfos to be equal.

Can you give a parametrically polymorphic definition for equality?

$$(\equiv) \quad :: \forall \alpha . \alpha \to \alpha \to \mathsf{Bool}$$
$$x \equiv y = ???$$

No. It's theoretically impossible.

Defining equality for specific datatypes

$$
\begin{array}{llll}
\mathbf{data}\ \mathsf{TimeInfo} & & & = AM \mid PM \mid H24 \\
(\equiv)_{\mathsf{TimeInfo}} & & & :: \mathsf{TimeInfo} \to \mathsf{TimeInfo} \to \mathsf{Bool} \\
AM & \equiv_{\mathsf{TimeInfo}} & AM & = True \\
PM & \equiv_{\mathsf{TimeInfo}} & PM & = True \\
H24 & \equiv_{\mathsf{TimeInfo}} & H24 & = True \\
\_ & \equiv_{\mathsf{TimeInfo}} & \_ & = False
\end{array}
$$

Defining equality for specific datatypes

$$
\begin{array}{llll}
\textbf{data}\ \mathsf{TimeInfo} & & & = AM \mid PM \mid H24 \\
(\equiv)_{\mathsf{TimeInfo}} & & & :: \mathsf{TimeInfo} \rightarrow \mathsf{TimeInfo} \rightarrow \mathsf{Bool} \\
AM & \equiv_{\mathsf{TimeInfo}} & AM & = True \\
PM & \equiv_{\mathsf{TimeInfo}} & PM & = True \\
H24 & \equiv_{\mathsf{TimeInfo}} & H24 & = True \\
\_ & \equiv_{\mathsf{TimeInfo}} & \_ & = False
\end{array}
$$

$$
\begin{array}{l}
\textbf{data}\ \mathsf{PackageDesc} = PD\ \mathsf{String\ Author\ Version\ Date} \\
(\equiv)_{\mathsf{PackageDesc}} :: \mathsf{PackageDesc} \rightarrow \mathsf{PackageDesc} \rightarrow \mathsf{Bool} \\
(PD\ name\ author\ version\ date) \equiv_{\mathsf{PackageDesc}} (PD\ name'\ author'\ version'\ date') \\
\quad = \begin{array}{llll}
 & name & \equiv_{\mathsf{String}} & name' \\
\wedge & author & \equiv_{\mathsf{Author}} & author' \\
\wedge & version & \equiv_{\mathsf{Version}} & version' \\
\wedge & date & \equiv_{\mathsf{Date}} & date'
\end{array}
\end{array}
$$

Defining equality for specific datatypes

$$
\begin{array}{llll}
\textbf{data}\ \mathsf{TimeInfo} & & & = AM \mid PM \mid H24 \\
(\equiv)_{\mathsf{TimeInfo}} & & & :: \mathsf{TimeInfo} \to \mathsf{TimeInfo} \to \mathsf{Bool} \\
AM & \equiv_{\mathsf{TimeInfo}} & AM & = True \\
PM & \equiv_{\mathsf{TimeInfo}} & PM & = True \\
H24 & \equiv_{\mathsf{TimeInfo}} & H24 & = True \\
\_ & \equiv_{\mathsf{TimeInfo}} & \_ & = False
\end{array}
$$

$$
\begin{array}{l}
\textbf{data}\ \mathsf{PackageDesc} = PD\ \mathsf{String}\ \mathsf{Author}\ \mathsf{Version}\ \mathsf{Date} \\
(\equiv)_{\mathsf{PackageDesc}} :: \mathsf{PackageDesc} \to \mathsf{PackageDesc} \to \mathsf{Bool} \\
(PD\ name\ author\ version\ date) \equiv_{\mathsf{PackageDesc}} (PD\ name'\ author'\ version'\ date') \\
\quad = \begin{array}{lll}
 & name & \equiv_{\mathsf{String}}\ name' \\
\wedge & author & \equiv_{\mathsf{Author}}\ author' \\
\wedge & version & \equiv_{\mathsf{Version}}\ version' \\
\wedge & date & \equiv_{\mathsf{Date}}\ date'
\end{array}
\end{array}
$$

$$
\textbf{data}\ \mathsf{Package} = P\ \mathsf{PackageDesc}\ [\mathsf{Package}]
$$

Lifting equality to parametrized datatypes

$$
\begin{array}{l}
\mathbf{data}\ \mathit{Maybe}\ \alpha = \mathit{Nothing} \mid \mathit{Just}\ \alpha \\
(\equiv)_{\mathsf{Maybe}} :: \forall \alpha.(\alpha \to \alpha \to \mathsf{Bool}) \to (\mathsf{Maybe}\ \alpha \to \mathsf{Maybe}\ \alpha \to \mathsf{Bool}) \\
(\equiv)_{\mathsf{Maybe}}\ (\equiv)_{\alpha}\ \mathit{Nothing}\ \mathit{Nothing} = \mathit{True} \\
(\equiv)_{\mathsf{Maybe}}\ (\equiv)_{\alpha}\ (\mathit{Just}\ x)\ \ (\mathit{Just}\ x') = x \equiv_{\alpha} x' \\
(\equiv)_{\mathsf{Maybe}}\ (\equiv)_{\alpha}\ \_\ \ \_ = \mathit{False}
\end{array}
$$

Lifting equality to parametrized datatypes

$$
\begin{array}{l}
\textbf{data}\ \mathit{Maybe}\ \alpha = \mathit{Nothing} \mid \mathit{Just}\ \alpha \\
(\equiv)_{\mathsf{Maybe}} :: \forall\alpha.(\alpha \to \alpha \to \mathsf{Bool}) \to (\mathsf{Maybe}\ \alpha \to \mathsf{Maybe}\ \alpha \to \mathsf{Bool}) \\
(\equiv)_{\mathsf{Maybe}}\ (\equiv)_{\alpha}\ \mathit{Nothing}\ \mathit{Nothing} = \mathit{True} \\
(\equiv)_{\mathsf{Maybe}}\ (\equiv)_{\alpha}\ (\mathit{Just}\ x)\ \ (\mathit{Just}\ x') = x \equiv_{\alpha} x' \\
(\equiv)_{\mathsf{Maybe}}\ (\equiv)_{\alpha}\ \_\ \ \_ = \mathit{False}
\end{array}
$$

$$
\begin{array}{l}
\textbf{data}\ [\alpha] = [\,] \mid \alpha : [\alpha] \\
(\equiv)_{[\,]} :: \forall\alpha.(\alpha \to \alpha \to \mathsf{Bool}) \to ([\alpha] \to [\alpha] \to \mathsf{Bool}) \\
(\equiv)_{[\,]}\ (\equiv)_{\alpha}\ [\,]\ \ [\,] = \mathit{True} \\
(\equiv)_{[\,]}\ (\equiv)_{\alpha}\ (x : xs)\ (x' : xs') = \quad x \equiv_{\alpha} x' \\
\qquad\qquad \wedge\ xs \equiv_{[\alpha]} xs' \\
\qquad\qquad \textbf{where}\ (\equiv)_{[\alpha]} = (\equiv)_{[\,]}\ (\equiv)_{\alpha} \\
(\equiv)_{[\,]}\ (\equiv)_{\alpha}\ \_\ \_ = \mathit{False}
\end{array}
$$

Abstraction, application and recursion

$$
\begin{array}{l}
\textbf{data}\ \mathsf{Package} = P\ \mathsf{PackageDesc}\ [\mathsf{Package}] \\
(\equiv)_{\mathsf{Package}} :: \mathsf{Package} \to \mathsf{Package} \to \mathsf{Bool} \\
(P\ \mathit{desc}\ \mathit{deps}) \equiv_{\mathsf{Package}} (P\ \mathit{desc}'\ \mathit{deps}') = \quad \mathit{desc} \equiv_{\mathsf{Package}} \mathit{desc}' \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ \mathit{deps} \equiv_{[\alpha]} \mathit{deps}' \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{where}\ (\equiv)_{[\alpha]} = (\equiv)_{[\,]}\ (\equiv)_{\alpha}
\end{array}
$$

Abstraction, application and recursion in the datatypes reappear in the definition of equality as abstraction, application and recursion on the value level!

Overloading

Haskell allows to place functions that work on different types into a type class:

$$\textbf{class}\ Eq\ \alpha\ \textbf{where}$$
$$(\equiv) \quad :: \alpha \to \alpha \to \mathsf{Bool}$$

Overloading

Haskell allows to place functions that work on different types into a type class:

$$
\begin{array}{l}
\textbf{class}\ Eq\ \alpha\ \textbf{where} \\
\quad (\equiv) \qquad\qquad :: \alpha \rightarrow \alpha \rightarrow \mathsf{Bool}
\end{array}
$$

We can make the previously defined functions **instances** of this class:

$$
\begin{array}{l}
\textbf{instance}\ Eq\ \mathsf{PackageDesc}\ \textbf{where} \\
\quad (\equiv) \qquad\qquad = (\equiv)_{\mathsf{PackageDesc}} \\
\textbf{instance}\ Eq\ \mathsf{Package}\ \textbf{where} \\
\quad (\equiv) \qquad\qquad = (\equiv)_{\mathsf{Package}} \\
\textbf{instance}\ Eq\ \alpha \Rightarrow Eq\ [\alpha]\ \textbf{where} \\
\quad (\equiv) \qquad\qquad = (\equiv)_{[\,]}\ (\equiv)
\end{array}
$$

Overloading

Haskell allows to place functions that work on different types into a type class:

$$
\begin{array}{l}
\textbf{class } Eq\ \alpha\ \textbf{where} \\
\quad (\equiv) \qquad\qquad :: \alpha \to \alpha \to \mathsf{Bool}
\end{array}
$$

We can make the previously defined functions **instances** of this class:

$$
\begin{array}{l}
\textbf{instance } Eq\ \mathsf{PackageDesc}\ \textbf{where} \\
\quad (\equiv) \qquad\qquad = (\equiv)_{\mathsf{PackageDesc}} \\
\textbf{instance } Eq\ \mathsf{Package}\ \textbf{where} \\
\quad (\equiv) \qquad\qquad = (\equiv)_{\mathsf{Package}} \\
\textbf{instance } Eq\ \alpha \Rightarrow Eq\ [\alpha]\ \textbf{where} \\
\quad (\equiv) \qquad\qquad = (\equiv)_{[\,]}\ (\equiv)
\end{array}
$$

The explicit argument of the equality function on a parametrized datatype is replaced by an implicit **dependency** $(Eq\ \alpha \Rightarrow)$ on the instance.

Overloading

Haskell allows to place functions that work on different types into a type class:

$$\textbf{class}\ \mathit{Eq}\ \alpha\ \textbf{where}$$
$$\quad (\equiv)_{\alpha} \quad :: \alpha \to \alpha \to \mathsf{Bool}$$

We can make the previously defined functions **instances** of this class:

$$\textbf{instance}\ \mathit{Eq}\ \mathsf{PackageDesc}\ \textbf{where}$$
$$\quad (\equiv)_{\mathsf{PackageDesc}} = (\equiv)_{\mathsf{PackageDesc}}$$
$$\textbf{instance}\ \mathit{Eq}\ \mathsf{Package}\ \textbf{where}$$
$$\quad (\equiv)_{\mathsf{Package}} = (\equiv)_{\mathsf{Package}}$$
$$\textbf{instance}\ \mathit{Eq}\ \alpha \Rightarrow \mathit{Eq}\ [\alpha]\ \textbf{where}$$
$$\quad (\equiv)_{[\,]} = (\equiv)_{[\,]}\ (\equiv)_{\alpha}$$

The explicit argument of the equality function on a parametrized datatype is replaced by an implicit **dependency** ($\mathit{Eq}\ \alpha \Rightarrow$) on the instance.

Is this satisfactory?

- Although we can use an overloaded version of equality on several datatypes now, we still had to define all the instance ourselves.
- Even worse, once we want to use equality on more datatypes, we have to define new instances.

Is this satisfactory?

- Although we can use an overloaded version of equality on several datatypes now, we still had to define all the instance ourselves.
- Even worse, once we want to use equality on more datatypes, we have to define new instances.
- On the other hand, it seems pretty obvious by now how to define equality for new datatypes:
  - It depends on the **structure** of the datatypes.

Is this satisfactory?

- Although we can use an overloaded version of equality on several datatypes now, we still had to define all the instance ourselves.
- Even worse, once we want to use equality on more datatypes, we have to define new instances.
- On the other hand, it seems pretty obvious by now how to define equality for new datatypes:
  - It depends on the **structure** of the datatypes.
  - Both values must belong to the same alternative.
  - All fields must be equal.
  - Abstraction, application and recursion must be handled in a natural way.

Is this satisfactory?

- Although we can use an overloaded version of equality on several datatypes now, we still had to define all the instance ourselves.
- Even worse, once we want to use equality on more datatypes, we have to define new instances.
- On the other hand, it seems pretty obvious by now how to define equality for new datatypes:
  - It depends on the **structure** of the datatypes.
  - Both values must belong to the same alternative.
  - All fields must be equal.
  - Abstraction, application and recursion must be handled in a natural way.

Generic programming makes the structure of datatypes available for the definition of type-dependent/type-indexed functions!

Generic programming in context

Ad-hoc polymorphism $\approx$ overloading

Structural polymorphism $\approx$ genericity

Parametric polymorphism

Haskell as a builtin **deriving** construct to magically derive functions such as $(\equiv)$, but this is only possible for a fixed amount of functions!

Overview

- Why Generic Haskell?
- Genericity and other types of polymorphism
- Writing and using generic functions in Generic Haskell
- Examples

Three datatypes

How does Generic Haskell expose the structure of datatypes?

Three datatypes

How does Generic Haskell expose the structure of datatypes?

It “deconstructs” datatypes so that they appear to built of a small set of relatively simple types.

Three datatypes

How does Generic Haskell expose the structure of datatypes?

It "deconstructs" datatypes so that they appear to built of a small set of relatively simple types.

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

- A value of Unit type represents a constructor with no fields (such as *Nothing* or the empty list).
- A Sum represents the choice between two alternatives.
- A Prod represents the sequence of two fields.

Generic functions

A function that is defined for the Unit, Sum, and Prod types is "generic".

- It works for all datatypes that do not contain primitive types.
- A **primitive** type is a datatype that can not be deconstructed because its implementation is hidden or because it cannot be defined by means of the Haskell **data** construct.
- Integers Int, characters Char, functions $(\rightarrow)$, and the IO monad are examples of primitive types.
- A generic function can also handle types containing primitive types, but then additional cases are needed for these primitive types.

From logical to functional programs

$$
\begin{array}{ll}
\textbf{class}\ \mathit{Eq}\ \alpha\ \textbf{where} & \\
\quad (\equiv)_{\alpha} & :: \alpha \rightarrow \alpha \rightarrow \mathsf{Bool} \\
\textbf{instance}\ \mathit{Eq}\ \mathsf{PackageDesc}\ \textbf{where} & \\
\quad (\equiv)_{\mathsf{PackageDesc}} & = (\equiv)_{\mathsf{PackageDesc}} \\
\textbf{instance}\ \mathit{Eq}\ \mathsf{Package}\ \textbf{where} & \\
\quad (\equiv)_{\mathsf{Package}} & = (\equiv)_{\mathsf{Package}} \\
\textbf{instance}\ \mathit{Eq}\ \alpha \Rightarrow \mathit{Eq}\ [\alpha]\ \textbf{where} & \\
\quad (\equiv)_{[\,]} & = (\equiv)_{[\,]}\ (\equiv)_{\alpha}
\end{array}
$$

Let us change the view …

From logical to functional programs

$$
\begin{array}{ll}
\textbf{class}\ \mathit{Eq}\ \alpha\ \textbf{where} & \\
\quad (\equiv)_{\alpha} & :: \alpha \to \alpha \to \mathsf{Bool} \\
\textbf{instance}\ \mathit{Eq}\ \mathsf{PackageDesc}\ \textbf{where} & \\
\quad (\equiv)_{\mathsf{PackageDesc}} & = (\equiv)_{\mathsf{PackageDesc}} \\
\textbf{instance}\ \mathit{Eq}\ \mathsf{Package}\ \textbf{where} & \\
\quad (\equiv)_{\mathsf{Package}} & = (\equiv)_{\mathsf{Package}} \\
\textbf{instance}\ \mathit{Eq}\ \alpha \Rightarrow \mathit{Eq}\ [\alpha]\ \textbf{where} & \\
\quad (\equiv)_{[\,]} & = (\equiv)_{[\,]}\ (\equiv)_{\alpha}
\end{array}
$$

Let us change the view ...

$$
\begin{array}{ll}
(\equiv)\ \langle \mathsf{PackageDesc} \rangle & = (\equiv)_{\mathsf{PackageDesc}} \\
(\equiv)\ \langle \mathsf{Package} \rangle & = (\equiv)_{\mathsf{Package}} \\
(\equiv)\ \langle [\alpha] \rangle & = (\equiv)_{[\,]}\ ((\equiv)\ \langle \alpha \rangle)
\end{array}
$$

This **type-indexed function** expresses the same as the instances above. Generic Haskell lets you define type-indexed functions.

Generic equality

$$
\begin{array}{lllll}
(\equiv)\ \langle \alpha \rangle & & & :: & \alpha \to \alpha \to \mathsf{Bool} \\
(\equiv)\ \langle \mathsf{Unit} \rangle & \mathit{Unit} & \mathit{Unit} & = & \mathit{True} \\
(\equiv)\ \langle \mathsf{Sum}\ \alpha\ \beta \rangle & (\mathit{Inl}\ x) & (\mathit{Inl}\ x') & = & (\equiv)\ \langle \alpha \rangle\ x\ x' \\
(\equiv)\ \langle \mathsf{Sum}\ \alpha\ \beta \rangle & (\mathit{Inr}\ y) & (\mathit{Inr}\ y') & = & (\equiv)\ \langle \alpha \rangle\ y\ y' \\
(\equiv)\ \langle \mathsf{Sum}\ \alpha\ \beta \rangle & \_ & \_ & = & \mathit{False} \\
(\equiv)\ \langle \mathsf{Prod}\ \alpha\ \beta \rangle & (x \times y) & (x' \times y') & = & (\equiv)\ \langle \alpha \rangle\ x\ x' \wedge (\equiv)\ \langle \beta \rangle\ y\ y' \\
(\equiv)\ \langle \mathsf{Int} \rangle & x & x' & = & (\equiv)_{\mathsf{Int}}\ x\ x' \\
(\equiv)\ \langle \mathsf{Char} \rangle & x & x' & = & (\equiv)_{\mathsf{Char}}\ x\ x'
\end{array}
$$

Generic equality

$$
\begin{array}{llll}
(\equiv)\ \langle\alpha\rangle & & & ::\ \alpha \to \alpha \to \mathsf{Bool} \\
(\equiv)\ \langle\mathsf{Unit}\rangle & \mathit{Unit} & \mathit{Unit} & = \mathit{True} \\
(\equiv)\ \langle\mathsf{Sum}\ \alpha\ \beta\rangle & (\mathit{Inl}\ x) & (\mathit{Inl}\ x') & = (\equiv)\ \langle\alpha\rangle\ x\ x' \\
(\equiv)\ \langle\mathsf{Sum}\ \alpha\ \beta\rangle & (\mathit{Inr}\ y) & (\mathit{Inr}\ y') & = (\equiv)\ \langle\alpha\rangle\ y\ y' \\
(\equiv)\ \langle\mathsf{Sum}\ \alpha\ \beta\rangle & \_ & \_ & = \mathit{False} \\
(\equiv)\ \langle\mathsf{Prod}\ \alpha\ \beta\rangle & (x \times y) & (x' \times y') & = (\equiv)\ \langle\alpha\rangle\ x\ x' \wedge (\equiv)\ \langle\beta\rangle\ y\ y' \\
(\equiv)\ \langle\mathsf{Int}\rangle & x & x' & = (\equiv)_{\mathsf{Int}}\ x\ x' \\
(\equiv)\ \langle\mathsf{Char}\rangle & x & x' & = (\equiv)_{\mathsf{Char}}\ x\ x'
\end{array}
$$

Additional cases such as

$$
\begin{array}{ll}
(\equiv)\ \langle\mathsf{PackageDesc}\rangle & = (\equiv)_{\mathsf{PackageDesc}} \\
(\equiv)\ \langle\mathsf{Package}\rangle & = (\equiv)_{\mathsf{Package}} \\
(\equiv)\ \langle[\alpha]\rangle & = (\equiv)_{[\,]}\ ((\equiv)\ \langle\alpha\rangle)
\end{array}
$$

are now superfluous. They are **implied** by the generic cases above.

Use of generic equality

The thus defined function can now be used on different datatypes.

$$\begin{array}{ll}
\textbf{data}\ \mathsf{TimeInfo} & = \mathit{AM} \mid \mathit{PM} \mid \mathit{H24} \\
\textbf{data}\ \mathsf{Tree}\ \alpha & = \mathit{Leaf}\ \alpha \mid \mathit{Node}\ (\mathsf{Tree}\ \alpha)\ (\mathsf{Tree}\ \alpha)
\end{array}$$

$$\begin{array}{lll}
(\equiv)\ \langle\mathsf{TimeInfo}\rangle & \mathit{AM}\ \mathit{H24} & \rightsquigarrow \mathit{False} \\
(\equiv)\ \langle\mathsf{TimeInfo}\rangle & \mathit{PM}\ \mathit{PM} & \rightsquigarrow \mathit{True} \\
(\equiv)\ \langle\mathsf{Tree\ Int}\rangle & (\mathit{Node}\ (\mathit{Node}\ (\mathit{Leaf}\ 2)\ (\mathit{Leaf}\ 4)) & \\
& \qquad\quad (\mathit{Node}\ (\mathit{Leaf}\ 1)\ (\mathit{Leaf}\ 3))) & \\
& (\mathit{Node}\ (\mathit{Node}\ (\mathit{Leaf}\ 4)\ (\mathit{Leaf}\ 2)) & \\
& \qquad\quad (\mathit{Node}\ (\mathit{Leaf}\ 1)\ (\mathit{Leaf}\ 3))) & \\
& & \rightsquigarrow \mathit{False}
\end{array}$$

What if we are only interested in the shape of the tree, not the values?

Local redefinition

Generic Haskell allows to locally redefine the generic function:

$$
\begin{aligned}
&\textbf{let}\ (\equiv)\ \langle \tau \rangle\ x\ x' = \mathit{True} \\
&\textbf{in}\ \ (\equiv)\ \langle \mathsf{Tree}\ \tau \rangle\ (\mathit{Node}\ (\mathit{Node}\ (\mathit{Leaf}\ 2)\ (\mathit{Leaf}\ 4)) \\
&\qquad\qquad\qquad\qquad\quad (\mathit{Node}\ (\mathit{Leaf}\ 1)\ (\mathit{Leaf}\ 3))) \\
&\qquad\qquad\qquad\quad (\mathit{Node}\ (\mathit{Node}\ (\mathit{Leaf}\ 4)\ (\mathit{Leaf}\ 2)) \\
&\qquad\qquad\qquad\qquad\quad (\mathit{Node}\ (\mathit{Leaf}\ 1)\ (\mathit{Leaf}\ 3))) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \rightsquigarrow \mathit{True}
\end{aligned}
$$

Here we have given a **name** (τ) to a position in the type and have redefined the behaviour of $(\equiv)$ for that position.

Generic abstraction

Generic Haskell allows to abstract common patterns of application for generic functions into new generic functions:

$$\begin{aligned} &\mathit{shapeequal}\ \langle\varphi\rangle = \mathbf{let}\ (\equiv)\ \langle\tau\rangle\ x\ x' = \mathit{True} \\ &\qquad\qquad\qquad\quad \mathbf{in}\ \ (\equiv)\ \langle\varphi\ \tau\rangle \end{aligned}$$

Now, *shapeequal* can be used for all type constructors, for instance for lists:

$$\begin{aligned} &\mathit{shapeequal}\ \langle[\,]\rangle\ [1,2,3]\ [4,5,6,7] \rightsquigarrow \mathit{False} \\ &\mathit{shapeequal}\ \langle[\,]\rangle\ [1,2,3]\ [4,5,6] \quad\ \rightsquigarrow \mathit{True} \end{aligned}$$

Generic functions, specific behaviour

- Sometimes, the automatically derived variant of a function for a specific datatype does not have the intended behaviour or is unnecessarily inefficient.
- A local redefinition might help here, but there is a simpler way.
- One can simply define a specific case for this datatype that overrides the generic definition.
- For instance, we could have defined the following specific case for equality on Packages if we know that a package is already uniquely defined in our application by its description:

$$
\begin{array}{l}
\textbf{data}\ \mathsf{PackageDesc} = \mathit{PD}\ \mathsf{String}\ \mathsf{Author}\ \mathsf{Version}\ \mathsf{Date} \\
\textbf{data}\ \mathsf{Package} \qquad\;\; = \mathit{P}\ \mathsf{PackageDesc}\ [\mathsf{Package}] \\
\ldots \\
(\equiv)\ \langle \mathsf{Package} \rangle\ (P\ \mathit{desc}\ \mathit{deps})\ (P\ \mathit{desc}'\ \mathit{deps}') = (\equiv)\ \langle \mathsf{PackageDesc} \rangle\ \mathit{desc}\ \mathit{desc}'
\end{array}
$$

Advantages of generic functions

- A generic function is written once, and works for a large class of datatypes.
- General algorithmic ideas that work for all datatypes can be expressed.
- The generic function itself can be typechecked. If the generic function is type correct, then so is every instance.
  This is different from meta-programming or programming with templates: although specific instances will be typechecked, the template or meta-program itself never is.
- There are complex datatypes for which the generic function is actually shorter and easier to write than the specific instance.

Overview

- Why Generic Haskell?
- Genericity and other types of polymorphism
- Writing and using generic functions in Generic Haskell
- Examples

Parsing and printing

Many forms of parsing and printing functions can be written generically. A very simple example is a function to encode a value as a list of Bits:

$$
\begin{array}{llll}
\textbf{data}\ \mathsf{Bit} = O \mid I \\
encode\ \langle\alpha\rangle & & :: \alpha \to [\mathsf{Bit}] \\
encode\ \langle\mathsf{Unit}\rangle & Unit & = [\,] \\
encode\ \langle\mathsf{Sum}\ \alpha\ \beta\rangle & (Inl\ x) & = O : encode\ \langle\alpha\rangle\ x \\
encode\ \langle\mathsf{Sum}\ \alpha\ \beta\rangle & (Inr\ y) & = I\ \ : encode\ \langle\beta\rangle\ y \\
encode\ \langle\mathsf{Prod}\ \alpha\ \beta\rangle & (x \times y) & = encode\ \langle\alpha\rangle\ x +\!\!+\ encode\ \langle\beta\rangle\ y \\
encode\ \langle\mathsf{Int}\rangle & x & = encodeInBits\ 32\ x \\
encode\ \langle\mathsf{Char}\rangle & x & = encodeInBits\ \ 8\ (ord\ x)
\end{array}
$$

$$
\begin{array}{ll}
\textbf{data}\ \mathsf{Tree}\ \alpha & = Leaf\ \alpha \mid Node\ (\mathsf{Tree}\ \alpha)\ (\mathsf{Tree}\ \alpha) \\
\textbf{data}\ \mathsf{TimeInfo} & = AM \mid PM \mid H24
\end{array}
$$

$$
\begin{array}{lll}
encode\ \langle\mathsf{TimeInfo}\rangle & H24 & \rightsquigarrow [I, I] \\
encode\ \langle\mathsf{Tree\ TimeInfo}\rangle & (Node\ (Leaf\ AM)\ (Leaf\ PM)) & \rightsquigarrow [I, O, O, O, I, O]
\end{array}
$$

Another generic function

$$
\begin{array}{llll}
\mathit{collect}\ \langle \alpha \rangle & & :: & \forall \rho . \alpha \to [\rho] \\
\mathit{collect}\ \langle \mathsf{Unit} \rangle & \mathit{Unit} & = & [\,] \\
\mathit{collect}\ \langle \mathsf{Sum}\ \alpha\ \beta \rangle & (\mathit{Inl}\ x) & = & \mathit{collect}\ \langle \alpha \rangle\ x \\
\mathit{collect}\ \langle \mathsf{Sum}\ \alpha\ \beta \rangle & (\mathit{Inr}\ y) & = & \mathit{collect}\ \langle \beta \rangle\ y \\
\mathit{collect}\ \langle \mathsf{Prod}\ \alpha\ \beta \rangle & (x \times y) & = & \mathit{collect}\ \langle \alpha \rangle\ x +\!\!+ \mathit{collect}\ \langle \beta \rangle\ y \\
\mathit{collect}\ \langle \mathsf{Int} \rangle & x & = & [\,] \\
\mathit{collect}\ \langle \mathsf{Char} \rangle & x & = & [\,]
\end{array}
$$

Another generic function

$$
\begin{array}{llll}
\mathit{collect}\ \langle\alpha\rangle & & :: & \forall\rho.\alpha \to [\rho] \\
\mathit{collect}\ \langle\mathsf{Unit}\rangle & \mathit{Unit} & = & [\,] \\
\mathit{collect}\ \langle\mathsf{Sum}\ \alpha\ \beta\rangle & (\mathit{Inl}\ x) & = & \mathit{collect}\ \langle\alpha\rangle\ x \\
\mathit{collect}\ \langle\mathsf{Sum}\ \alpha\ \beta\rangle & (\mathit{Inr}\ y) & = & \mathit{collect}\ \langle\beta\rangle\ y \\
\mathit{collect}\ \langle\mathsf{Prod}\ \alpha\ \beta\rangle & (x \times y) & = & \mathit{collect}\ \langle\alpha\rangle\ x +\!\!+\ \mathit{collect}\ \langle\beta\rangle\ y \\
\mathit{collect}\ \langle\mathsf{Int}\rangle & x & = & [\,] \\
\mathit{collect}\ \langle\mathsf{Char}\rangle & x & = & [\,]
\end{array}
$$

- Alone, this generic function is completely useless! It **always** returns the empty list.
- The function *collect* is, however, a good basis for local redefinition or **extension**.
- Collect all elements from a tree:

$$
\begin{array}{l}
\mathbf{let}\ \mathit{collect}\ \langle\tau\rangle\ x = [x] \\
\mathbf{in}\ \ \mathit{collect}\ \langle\mathsf{Tree}\ \tau\rangle\ (\mathit{Node}\ (\mathit{Leaf}\ 1)\ (\mathit{Leaf}\ 2) \\
\qquad\qquad\qquad\qquad\qquad (\mathit{Leaf}\ 3)\ (\mathit{Leaf}\ 4)) \rightsquigarrow [1,2,3,4]
\end{array}
$$

Traversals

- With functions such as *collect* as a base, so-called generic **traversals** can be written.
- If the abstract syntax of a language is expressed as a system of datatypes, generic functions can be used to perform operations such as:
  - determine free variables in a part of a program
  - perform optimizations
  - perform modifications

Traversal example

$$
\begin{array}{ll}
\textbf{data}\ \mathit{Compiler} & = C\ \mathit{Name}\ [\mathit{Package}\ \mathit{Maintainer}] \\
\textbf{data}\ \mathit{Package}\ a & = P\ \mathit{Name}\ a\ [\mathit{Feature}]\ [\mathit{Package}\ a] \\
\textbf{data}\ \mathit{Maintainer} & = M\ \mathit{Name}\ \mathit{Affiliation} \\
 & \mid\ \mathit{Unmaintained} \\
\textbf{data}\ \mathit{Feature} & = F\ \mathit{String} \\
\textbf{type}\ \mathit{Name} & = \mathit{String} \\
\textbf{type}\ \mathit{Affiliation} & = \mathit{String}
\end{array}
$$

Possible tasks:

- Check if something is maintained.
- Assign a new maintainer to a structure.
- Assign all unmaintained packages that implement generic programming to me.

Summary of Generic Haskell

- Type-indexed functions can be defined that generically work for all datatypes.
- With generic abstraction, local redefinition, and extension there are several possibilities to build new functions from a library of basic generic functions.
- Generic functions can interact, i.e. depend on one another. In this talk we have mainly seen functions that are recursive.
- Datatypes can also be indexed by a type argument. Generic Haskell supports those as well.
- Applications range from classic functions such as equality over all kinds of printing, parsing, conversions, mappings, over generic traversals, selectively modying large trees, to operations on XML documents and the automatic derivation of isomorphisms between different datatypes.

Implementation of Generic Haskell

- Generic Haskell can be obtained from `www.generic-haskell.org`.
- It implements all the features presented in this talk, but with a slightly different syntax. (Generic Haskell is still in development and may change significantly between releases.)
- The Generic Haskell compiler translates generic functions into ordinary Haskell functions via specialisation: instances for concrete datatypes are computed and inserted at the appropriate positions.
- Type checking is left to the Haskell compiler, but if the Haskell file typechecks, all generic definitions are type correct.