Structural polymorphism in Generic Haskell

Andres Löh

andres@cs.uu.nl

5 February 2005

Universiteit Utrecht

Overview

About Haskell

Genericity and other types of polymorphism

Examples of generic functions

Generic Haskell

Overview

About Haskell

Genericity and other types of polymorphism

Examples of generic functions

Generic Haskell

Haskell

- Haskell is a statically typed, pure functional language with lazy evaluation.
- Functions are defined by pattern matching.

```
factorial 0 = 1
factorial n = n · factorial (n − 1)
```

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

```
factorial :: Int → Int
```

- Functions with multiple arguments are written in curried style.

```
and :: Bool → Bool → Bool
and True True = True
and _    _    = False
```

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.

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

- Functions with multiple arguments are written in curried style.

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

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.

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

- Functions with multiple arguments are written in curried style.

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

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.

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

- Functions with multiple arguments are written in curried style.

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

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.

$$\mathit{factorial} \quad :: \mathsf{Int} \to \mathsf{Int}$$

- Functions with multiple arguments are written in curried style.

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

User-defined datatypes

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

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

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

- Functions are often defined recursively, by induction on the structure of a datatype:

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

User-defined datatypes

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

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

  The expression $\mathit{Succ}\ (\mathit{Succ}\ (\mathit{Succ}\ \mathit{Zero}))$ represents the number 3.
- Functions are often defined recursively, by induction on the structure of a datatype:

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

Haskell datatypes

Haskell's **data** construct is extremely flexible.

```
data TimeInfo  = AM | PM | H24
data Package   = P String Author Version Date
data Maybe α   = Nothing | Just α
data [α]       = [] | α : [α]
data Tree α    = Leaf α | Node (Tree α) (Tree α)
```

Common structure:

- parametrized over a number of arguments
- several constructors / alternatives
- multiple fields per constructor
- possibly recursion

Haskell datatypes

Haskell's **data** construct is extremely flexible.

```
data TimeInfo = AM | PM | H24
data Package  = P String Author Version Date
data Maybe α  = Nothing | Just α
data [α]      = [] | α : [α]
data Tree α   = Leaf α | Node (Tree α) (Tree α)
```

Common structure:

- parametrized over a number of arguments
- several constructors / alternatives
- multiple fields per constructor
- possibly recursion

Haskell datatypes

Haskell's **data** construct is extremely flexible.

$$
\begin{array}{ll}
\textbf{data}\ \mathsf{TimeInfo} & = \mathit{AM} \mid \mathit{PM} \mid \mathit{H24} \\
\textbf{data}\ \mathsf{Package} & = \mathit{P}\ \mathsf{String}\ \mathsf{Author}\ \mathsf{Version}\ \mathsf{Date} \\
\textbf{data}\ \mathsf{Maybe}\ \alpha & = \mathit{Nothing} \mid \mathit{Just}\ \alpha \\
\textbf{data}\ [\alpha] & = [\,] \mid \alpha : [\alpha] \\
\textbf{data}\ \mathsf{Tree}\ \alpha & = \mathit{Leaf}\ \alpha \mid \mathit{Node}\ (\mathsf{Tree}\ \alpha)\ (\mathsf{Tree}\ \alpha)
\end{array}
$$

Common structure:

- parametrized over a number of arguments
- several constructors / alternatives
- multiple fields per constructor
- possibly recursion

Haskell datatypes

Haskell's **data** construct is extremely flexible.

$$
\begin{array}{lll}
\textbf{data}\ \mathsf{TimeInfo} & = & \mathit{AM} \mid \mathit{PM} \mid \mathit{H24} \\
\textbf{data}\ \mathsf{Package} & = & P\ \mathsf{String}\ \mathsf{Author}\ \mathsf{Version}\ \mathsf{Date} \\
\textbf{data}\ \mathsf{Maybe}\ \alpha & = & \mathit{Nothing} \mid \mathit{Just}\ \alpha \\
\textbf{data}\ [\alpha] & = & [\,] \mid \alpha : [\alpha] \\
\textbf{data}\ \mathsf{Tree}\ \alpha & = & \mathit{Leaf}\ \alpha \mid \mathit{Node}\ (\mathsf{Tree}\ \alpha)\ (\mathsf{Tree}\ \alpha)
\end{array}
$$

Common structure:

- parametrized over a number of arguments
- several constructors / alternatives
- multiple fields per constructor
- possibly recursion

Haskell datatypes

Haskell's **data** construct is extremely flexible.

$$
\begin{aligned}
&\textbf{data}\ \mathsf{TimeInfo} &&= \mathit{AM} \mid \mathit{PM} \mid \mathit{H24}\\
&\textbf{data}\ \mathsf{Package} &&= \mathit{P}\ \mathsf{String}\ \mathsf{Author}\ \mathsf{Version}\ \mathsf{Date}\\
&\textbf{data}\ \mathsf{Maybe}\ \alpha &&= \mathit{Nothing} \mid \mathit{Just}\ \alpha\\
&\textbf{data}\ [\alpha] &&= [\,] \mid \alpha : [\alpha]\\
&\textbf{data}\ \mathsf{Tree}\ \alpha &&= \mathit{Leaf}\ \alpha \mid \mathit{Node}\ (\mathsf{Tree}\ \alpha)\ (\mathsf{Tree}\ \alpha)
\end{aligned}
$$

Common structure:

- parametrized over a number of arguments
- several constructors / alternatives
- multiple fields per constructor
- possibly recursion

Haskell datatypes

Haskell's **data** construct is extremely flexible.

$$
\begin{aligned}
&\textbf{data}\ \mathsf{TimeInfo} &&= \mathit{AM} \mid \mathit{PM} \mid \mathit{H24}\\
&\textbf{data}\ \mathsf{Package} &&= \mathit{P}\ \mathsf{String}\ \mathsf{Author}\ \mathsf{Version}\ \mathsf{Date}\\
&\textbf{data}\ \mathsf{Maybe}\ \alpha &&= \mathit{Nothing} \mid \mathit{Just}\ \alpha\\
&\textbf{data}\ [\alpha] &&= [\,] \mid \alpha : [\alpha]\\
&\textbf{data}\ \mathsf{Tree}\ \alpha &&= \mathit{Leaf}\ \alpha \mid \mathit{Node}\ (\mathsf{Tree}\ \alpha)\ (\mathsf{Tree}\ \alpha)
\end{aligned}
$$

Common structure:

- parametrized over a number of arguments
- several constructors / alternatives
- multiple fields per constructor
- possibly recursion

Overview

About Haskell

Genericity and other types of polymorphism

Examples of generic functions

Generic Haskell

Parametric polymorphism

Haskell allows to express functions that work on all datatypes in a uniform way.

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

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

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

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

When are two values equal?

It is easy to define an equality function for a specific datatype.

- Both values must belong to the same alternative.
- The corresponding fields must be equal.

When are two values equal?

It is easy to define an equality function for a specific datatype.

- Both values must belong to the same alternative.
- The corresponding fields must be equal.

They must belong to the same alternative

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

The corresponding fields must be equal

data Package $= PD$ String Author Version Date

$(==)_{\text{Package}}$ :: Package $\rightarrow$ Package $\rightarrow$ Bool

$$
\begin{aligned}
(PD\ n\ c\ v\ d) ==_{\text{Package}} (PD\ n'\ c'\ v'\ d') = \quad & n ==_{\text{String}}\ n' \\
\wedge\ & c ==_{\text{Author}}\ c' \\
\wedge\ & v ==_{\text{Version}}\ v' \\
\wedge\ & d ==_{\text{Date}}\ d'
\end{aligned}
$$

Equality for parametrized datatypes

$$
\begin{array}{llll}
\multicolumn{4}{l}{\mathbf{data}\ \mathsf{Maybe}\ \alpha = \mathit{Nothing} \mid \mathit{Just}\ \alpha} \\
(\texttt{==})_{\mathsf{Maybe}} :: \forall\alpha. & & & \to (\mathsf{Maybe}\ \alpha \to \mathsf{Maybe}\ \alpha \to \mathsf{Bool}) \\
(\texttt{==})_{\mathsf{Maybe}} & \mathit{Nothing} & \mathit{Nothing} & = \mathit{True} \\
(\texttt{==})_{\mathsf{Maybe}} & (\mathit{Just}\ x) & (\mathit{Just}\ x') & = x \ \texttt{==}_{???}\ x' \\
(\texttt{==})_{\mathsf{Maybe}} & \_ & \_ & = \mathit{False}
\end{array}
$$

- We can define the equality for parametrized datatypes, but for that, we must know the equality function(s) for the argument(s).
- The equality function depends on itself.

Equality for parametrized datatypes

$$
\begin{aligned}
&\textbf{data}\ \mathsf{Maybe}\ \alpha = \mathit{Nothing} \mid \mathit{Just}\ \alpha \\
&(\mathtt{==})_{\mathsf{Maybe}} :: \forall\alpha.(\alpha \to \alpha \to \mathsf{Bool}) \to (\mathsf{Maybe}\ \alpha \to \mathsf{Maybe}\ \alpha \to \mathsf{Bool}) \\
&(\mathtt{==})_{\mathsf{Maybe}}\ (\mathtt{==})_{\alpha}\ \mathit{Nothing}\ \mathit{Nothing} = \mathit{True} \\
&(\mathtt{==})_{\mathsf{Maybe}}\ (\mathtt{==})_{\alpha}\ (\mathit{Just}\ x)\ (\mathit{Just}\ x') = x \mathrel{\mathtt{==}_{\alpha}} x' \\
&(\mathtt{==})_{\mathsf{Maybe}}\ (\mathtt{==})_{\alpha}\ \_\ \_ = \mathit{False}
\end{aligned}
$$

- We can define the equality for parametrized datatypes, but for that, we must know the equality function(s) for the argument(s).
- The equality function depends on itself.

Equality for parametrized datatypes

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

- We can define the equality for parametrized datatypes, but for that, we must know the equality function(s) for the argument(s).
- The equality function depends on itself.

Equality for parametrized datatypes

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

- We can define the equality for parametrized datatypes, but for that, we must know the equality function(s) for the argument(s).
- The equality function depends on itself.

Equality isn't parametrically polymorphic

- We know intuitively what it means for two Packages to be equal.
- We also know what it means for two Trees, Maybes or TimeInfos to be equal.
- However, it is impossible to give a parametrically polymorphic definition for equality:

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

- This is a consequence of the Parametricity Theorem (Reynolds 1983).

Equality isn't parametrically polymorphic

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

- However, it is impossible to give a parametrically polymorphic definition for equality:

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

- This is a consequence of the Parametricity Theorem (Reynolds 1983).

Overloading or ad-hoc polymorphism

- We have seen that we can define specific equality functions for many datatypes, following the intuitive algorithm that two values are equal iff
  - both values belong to the same alternative,
  - the corresponding fields are equal.
- A parametrically polymorphic equality function is impossible, because equality needs to access the structure of the datatypes to perform the comparison.
- Haskell allows to place functions that work on different types into a type class.
- Then, we can use the same name (==) for all the specific equality functions.

Overloading or ad-hoc polymorphism

- We have seen that we can define specific equality functions for many datatypes, following the intuitive algorithm that two values are equal iff
  - both values belong to the same alternative,
  - the corresponding fields are equal.
- A parametrically polymorphic equality function is impossible, because equality needs to access the structure of the datatypes to perform the comparison.
- Haskell allows to place functions that work on different types into a type class.
- Then, we can use the same name (==) for all the specific equality functions.

Overloading or ad-hoc polymorphism

- We have seen that we can define specific equality functions for many datatypes, following the intuitive algorithm that two values are equal iff
  - both values belong to the same alternative,
  - the corresponding fields are equal.
- A parametrically polymorphic equality function is impossible, because equality needs to access the structure of the datatypes to perform the comparison.
- Haskell allows to place functions that work on different types into a type class.
- Then, we can use the same name (==) for all the specific equality functions.

Overloading or ad-hoc polymorphism

- We have seen that we can define specific equality functions for many datatypes, following the intuitive algorithm that two values are equal iff
  - both values belong to the same alternative,
  - the corresponding fields are equal.
- A parametrically polymorphic equality function is impossible, because equality needs to access the structure of the datatypes to perform the comparison.
- Haskell allows to place functions that work on different types into a type class.
- Then, we can use the same name (==) for all the specific equality functions.

Type classes

A type class defines a set of datatypes that support common operations:

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

A type can be made an instance of the class by defining the class operations:

$$
\begin{array}{lll}
\textbf{instance}\ \mathit{Eq}\ \mathsf{TimeInfo} & \textbf{where}\ (\texttt{==}) & = (\texttt{==})_{\mathsf{TimeInfo}} \\
\textbf{instance}\ \mathit{Eq}\ \mathsf{Package} & \textbf{where}\ (\texttt{==}) & = (\texttt{==})_{\mathsf{Package}} \\
\textbf{instance}\ \mathit{Eq}\ \alpha \Rightarrow \mathit{Eq}\ [\alpha] & \textbf{where}\ (\texttt{==}) & = (\texttt{==})_{[]}\ (\texttt{==})
\end{array}
$$

The dependency of equality turns into an instance constraint.

Type classes

A type class defines a set of datatypes that support common operations:

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

A type can be made an instance of the class by defining the class operations:

$$\begin{array}{lll}
\textbf{instance}\ Eq\ \mathsf{TimeInfo} & \textbf{where}\ (\texttt{==}) & = (\texttt{==})_{\mathsf{TimeInfo}} \\
\textbf{instance}\ Eq\ \mathsf{Package} & \textbf{where}\ (\texttt{==}) & = (\texttt{==})_{\mathsf{Package}} \\
\textbf{instance}\ Eq\ \alpha \Rightarrow Eq\ [\alpha] & \textbf{where}\ (\texttt{==}) & = (\texttt{==})_{[\,]}\ (\texttt{==})
\end{array}$$

The dependency of equality turns into an instance constraint.

Type classes

A type class defines a set of datatypes that support common operations:

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

A type can be made an instance of the class by defining the class operations:

$$\begin{array}{lll}
\textbf{instance } Eq\ \mathsf{TimeInfo} & \textbf{where } (==) & = (==)_{\mathsf{TimeInfo}} \\
\textbf{instance } Eq\ \mathsf{Package} & \textbf{where } (==) & = (==)_{\mathsf{Package}} \\
\textbf{instance } Eq\ \alpha \Rightarrow Eq\ [\alpha] & \textbf{where } (==) & = (==)_{[\,]}\ (==)
\end{array}$$

The dependency of equality turns into an instance constraint.

Type classes

A type class defines a set of datatypes that support common operations:

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

A type can be made an instance of the class by defining the class operations:

$$\begin{array}{lll}
\textbf{instance}\ Eq\ \mathsf{TimeInfo} & \textbf{where}\ (==)_{\mathsf{TimeInfo}} & = (==)_{\mathsf{TimeInfo}} \\
\textbf{instance}\ Eq\ \mathsf{Package} & \textbf{where}\ (==)_{\mathsf{Package}} & = (==)_{\mathsf{Package}} \\
\textbf{instance}\ Eq\ \alpha \Rightarrow Eq\ [\alpha] & \textbf{where}\ (==)_{[\,]} & = (==)_{[\,]}\ (==)_{\alpha}
\end{array}$$

The dependency of equality turns into an instance constraint.

Is this satisfactory?

- We can use an overloaded version of equality on several datatypes now.
- We had to define all the instances ourselves, in an ad-hoc way.
- Once we want to use equality on more datatypes, we have to define new instances.

Let us define the equality function once and for all!

Is this satisfactory?

- We can use an overloaded version of equality on several datatypes now.
- We had to define all the instances ourselves, in an ad-hoc way.
- Once we want to use equality on more datatypes, we have to define new instances.

Let us define the equality function once and for all!

Structural polymorphism

Structural polymorphism (also called generic programming) makes the structure of datatypes available for the definition of type-indexed functions!

Generic programming in context

Ad-hoc polymorphism $\approx$ overloading

Parametric polymorphism

Generic programming in context

Ad-hoc polymorphism ≈ overloading

Structural polymorphism ≈ genericity

Parametric polymorphism

Overview

About Haskell

Genericity and other types of polymorphism

Examples of generic functions

Generic Haskell

Generic equality

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

$$
\begin{array}{l}
\mathbf{data}\ \mathsf{Unit} = \mathit{Unit} \\
\mathbf{data}\ \mathsf{Sum}\ \alpha\ \beta = \mathit{Inl}\ \alpha \mid \mathit{Inr}\ \beta \\
\mathbf{data}\ \mathsf{Prod}\ \alpha\ \beta = \alpha \times \beta
\end{array}
$$

Generic equality

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

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

Generic equality

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

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

Generic equality

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

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

Generic equality

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

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

Generic functions

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

- It works automatically for "all" datatypes.
- Datatypes are implicitly deconstructed into a representation that involves Unit, Sum, and Prod.
- Primitive or abstract types might require special cases in the definition.

Generic functions

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

- It works automatically for "all" datatypes.
- Datatypes are implicitly deconstructed into a representation that involves Unit, Sum, and Prod.
- Primitive or abstract types might require special cases in the definition.

Generic functions

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

- It works automatically for "all" datatypes.
- Datatypes are implicitly deconstructed into a representation that involves Unit, Sum, and Prod.
- Primitive or abstract types might require special cases in the definition.

Generic functions

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

- It works automatically for "all" datatypes.
- Datatypes are implicitly deconstructed into a representation that involves Unit, Sum, and Prod.
- Primitive or abstract types might require special cases in the definition.

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 (such as Int, Char, $(\rightarrow)$, and IO).
- If a generic function is supposed to work for types containing a primitive type, it has to be defined for this primitive type.

$$
\begin{aligned}
(==)\ \langle \mathsf{Int} \rangle\ x\ x' &= (==)_{\mathsf{Int}}\ x\ x' \\
(==)\ \langle \mathsf{Char} \rangle\ x\ x' &= (==)_{\mathsf{Char}}\ x\ x'
\end{aligned}
$$

- Abstract types, where the programmer specifically hides the implementation, are treated in the same way as primitive types.

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 (such as Int, Char, $(\rightarrow)$, and IO).
- If a generic function is supposed to work for types containing a primitive type, it has to be defined for this primitive type.

$$
\begin{aligned}
(==)\ \langle \mathsf{Int} \rangle\ \ x\ x' &= (==)_{\mathsf{Int}}\ \ x\ x' \\
(==)\ \langle \mathsf{Char} \rangle\ x\ x' &= (==)_{\mathsf{Char}}\ x\ x'
\end{aligned}
$$

- Abstract types, where the programmer specifically hides the implementation, are treated in the same way as primitive types.

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 (such as Int, Char, $(\to)$, and IO).
- If a generic function is supposed to work for types containing a primitive type, it has to be defined for this primitive type.

$$
\begin{aligned}
(==)\ \langle \mathsf{Int} \rangle\ x\ x' &= (==)_{\mathsf{Int}}\ x\ x' \\
(==)\ \langle \mathsf{Char} \rangle\ x\ x' &= (==)_{\mathsf{Char}}\ x\ x'
\end{aligned}
$$

- Abstract types, where the programmer specifically hides the implementation, are treated in the same way as primitive types.

Deconstruction into Unit, Sum, Prod

- 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.

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

$$
\begin{aligned}
&\mathbf{data}\ \mathsf{Tree}\ \alpha = \mathit{Leaf}\ \alpha \mid \mathit{Node}\ (\mathsf{Tree}\ \alpha)\ (\mathsf{Tree}\ \alpha) \\
&\mathsf{Tree}\ \alpha \approx \mathsf{Sum}\ \alpha\ (\mathsf{Prod}\ (\mathsf{Tree}\ \alpha)\ (\mathsf{Tree}\ \alpha))
\end{aligned}
$$

Deconstruction into Unit, Sum, Prod

- 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.

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

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

$$\mathsf{Tree}\ \alpha \approx \mathsf{Sum}\ \alpha\ (\mathsf{Prod}\ (\mathsf{Tree}\ \alpha)\ (\mathsf{Tree}\ \alpha))$$

Universiteit Utrecht

Using a generic function

The defined equality function can now be used at different datatypes.

$$
\begin{aligned}
&\mathbf{data}\ \mathsf{TimeInfo} = \mathit{AM} \mid \mathit{PM} \mid \mathit{H24} \\
&\mathbf{data}\ \mathsf{Tree}\ \alpha \quad\ = \mathit{Leaf}\ \alpha \mid \mathit{Node}\ (\mathsf{Tree}\ \alpha)\ (\mathsf{Tree}\ \alpha)
\end{aligned}
$$

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

Applications for generic functions

- comparison
  - equality
  - ordering
- parsing and printing
  - read/write a canonical representation
  - read/write a binary representation
  - read/write XML to/from a typed Haskell value
  - compression, encryption
- generation
  - generating default values
  - enumerating all values of a datatype
  - (random) generation of test data
- traversals
  - collecting and combining data from a tree
  - modifying data in a tree

Applications for generic functions

- comparison
  - equality
  - ordering
- parsing and printing
  - read/write a canonical representation
  - read/write a binary representation
  - read/write XML to/from a typed Haskell value
  - compression, encryption
- generation
  - generating default values
  - enumerating all values of a datatype
  - (random) generation of test data
- traversals
  - collecting and combining data from a tree
  - modifying data in a tree

Applications for generic functions

- comparison
  - equality
  - ordering
- parsing and printing
  - read/write a canonical representation
  - read/write a binary representation
  - read/write XML to/from a typed Haskell value
  - compression, encryption
- generation
  - generating default values
  - enumerating all values of a datatype
  - (random) generation of test data
- traversals
  - collecting and combining data from a tree
  - modifying data in a tree

Applications for generic functions

- comparison
  - equality
  - ordering
- parsing and printing
  - read/write a canonical representation
  - read/write a binary representation
  - read/write XML to/from a typed Haskell value
  - compression, encryption
- generation
  - generating default values
  - enumerating all values of a datatype
  - (random) generation of test data
- traversals
  - collecting and combining data from a tree
  - modifying data in a tree

Advantages of generic functions

- reusable
- type safe
- simple
- adaptable

Advantages of generic functions

- reusable
- type safe
- simple
- adaptable

Advantages of generic functions

- reusable
- type safe
- simple
- adaptable

Advantages of generic functions

- reusable
- type safe
- simple
- adaptable

Overview

About Haskell

Genericity and other types of polymorphism

Examples of generic functions

Generic Haskell

The Generic Haskell Project

- A research project funded by the NWO (Dutch Research Organisation) from October 2000 until October 2004.
- Goal: create a language extension for Haskell that supports generic programming.
- Based on earlier work by Johan Jeuring and Ralf Hinze.
- Project is now finished, but work on Generic Haskell will continue in Utrecht.
- Results: compared to the original ideas, much easier to use, yet more expressive.
- The PhD thesis "Exploring Generic Haskell" is a reasonably complete documentation of the results of the project.

The Generic Haskell Project

- A research project funded by the NWO (Dutch Research Organisation) from October 2000 until October 2004.
- Goal: create a language extension for Haskell that supports generic programming.
- Based on earlier work by Johan Jeuring and Ralf Hinze.
- Project is now finished, but work on Generic Haskell will continue in Utrecht.
- Results: compared to the original ideas, much easier to use, yet more expressive.
- The PhD thesis “Exploring Generic Haskell” is a reasonably complete documentation of the results of the project.

The Generic Haskell Compiler

- … is a preprocessor for the Haskell language.
- It extends Haskell with constructs to define
  - type-indexed functions (which can be generic),
  - type-indexed datatypes.
- Generic Haskell compiles datatypes of the input language to isomorphic structural representations using Unit, Sum, and Prod.
- Generic Haskell compiles generic functions to specialized functions that work for specific types.
- Generic Haskell compiles calls to generic functions into calls to the specialisations.

Additional features

- Several mechanisms to define new generic definitions out of existing ones:
  - local redefinition allows to change the behaviour of a generic function on a specific type locally
  - generic abstraction allows to define generic functions in terms of other generic functions without fixing the type argument
  - default cases allow to extend generic functions with additional cases for specific types

Additional features

- Several mechanisms to define new generic definitions out of existing ones:
  - local redefinition allows to change the behaviour of a generic function on a specific type locally
  - generic abstraction allows to define generic functions in terms of other generic functions without fixing the type argument
  - default cases allow to extend generic functions with additional cases for specific types

Additional features

- Several mechanisms to define new generic definitions out of existing ones:
  - local redefinition allows to change the behaviour of a generic function on a specific type locally
  - generic abstraction allows to define generic functions in terms of other generic functions without fixing the type argument
  - default cases allow to extend generic functions with additional cases for specific types

Dependencies

- Generic functions can interact, i.e., depend on one another.
- For instance, equality depends on itself.
- There are generic functions that depend on multiple other generic functions.
- Dependencies are tracked by the type system in Generic Haskell.

Type-indexed datatypes

- Generic functions are functions defined on the structure of datatypes.
- Type-indexed datatypes are datatypes defined on the structure of datatypes.
- Type-indexed tries are finite maps that employ the shape of the key datatype to store the values more efficiently.
- The zipper is a data structure that facilitates editing operations on a datatype.

Type-indexed datatypes

- Generic functions are functions defined on the structure of datatypes.
- Type-indexed datatypes are datatypes defined on the structure of datatypes.
- Type-indexed tries are finite maps that employ the shape of the key datatype to store the values more efficiently.
- The zipper is a data structure that facilitates editing operations on a datatype.

Type-indexed datatypes

- Generic functions are functions defined on the structure of datatypes.
- Type-indexed datatypes are datatypes defined on the structure of datatypes.
- Type-indexed tries are finite maps that employ the shape of the key datatype to store the values more efficiently.
- The zipper is a data structure that facilitates editing operations on a datatype.

Implementation of Generic Haskell

- Generic Haskell can be obtained from `www.generic-haskell.org`.
- The current release (from January 2005) should contain all the features mentioned in this talk (except for syntactical differences when using type-indexed types).

Related work

- Scrap your boilerplate (Lämmel and Peyton Jones)
- Pattern calculus (Jay)
- Dependently typed programming (Augustsson, Altenkirch and McBride, ...)
- Intensional type analysis (Harper, Morrisett, Weirich)
- GADTs (Cheney and Hinze, Weirich and Peyton Jones)
- Template Haskell (Sheard and Peyton Jones)
- Templates in C++
- Generics in C# and Java
- ...

Future work

- Generic views, i.e., different structural representations of datatypes for different sorts of applications.
- Type inference.
- First-class generic functions.
- …

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:

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

Parsing and printing – contd.

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

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

Traversals

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

Alone, this generic function is completely useless! It always returns the empty list.

Traversals

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

Alone, this generic function is completely useless! It always returns the empty list.

Local redefinition and *collect*

The function *collect* is a good basis for local redefinition.

Collect all elements from a tree:

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

Local redefinition

$$\begin{array}{l}\textbf{let}\ (\texttt{==})\ \langle\alpha\rangle\ x\ y = (\texttt{==})\ \langle\mathsf{Char}\rangle\ (\mathit{toUpper}\ x)\ (\mathit{toUpper}\ y)\\ \textbf{in}\ \ (\texttt{==})\ \langle[\alpha]\rangle\ \texttt{"generic Haskell"}\ \texttt{"Generic HASKELL"}\end{array}$$

Generic abstraction

> $symmetric\ \langle\alpha\rangle\ x = equal\ \langle\alpha\rangle\ x\ (reverse\ \langle\alpha\rangle\ x)$

Universiteit Utrecht

Type-indexed tries

$$
\begin{array}{lll}
\textbf{type}\ \mathsf{FMap}\ \langle\mathsf{Unit}\rangle & \mathit{val} = & \mathsf{Maybe}\ \mathit{val} \\
\textbf{type}\ \mathsf{FMap}\ \langle\mathsf{Sum}\ \alpha\ \beta\rangle & \mathit{val} = & (\mathsf{FMap}\ \langle\alpha\rangle\ \mathit{val}, \mathsf{FMap}\ \langle\beta\rangle\ \mathit{val}) \\
\textbf{type}\ \mathsf{FMap}\ \langle\mathsf{Prod}\ \alpha\ \beta\rangle & \mathit{val} = & \mathsf{FMap}\ \langle\alpha\rangle\ (\mathsf{FMap}\ \langle\beta\rangle\ \mathit{val})
\end{array}
$$

Universiteit Utrecht