Generalizing Generic Functions

Andres Löh

7 July 2004
Motivation

Despite “dependency style” Generic Haskell, generic functions have a number of restrictions:

▶ only one type argument
▶ no higher-order type-indexed functions
▶ only flat type patterns
▶ complicated types for generic functions of higher arity
▶ no inference of type arguments
Motivation

Despite “dependency style” Generic Haskell, generic functions have a number of restrictions:

▶ only one type argument
▶ no higher-order type-indexed functions
▶ only flat type patterns
▶ complicated types for generic functions of higher arity
▶ no inference of type arguments

Not all of the restrictions pose difficult problems, but all of them are “remaining work”.
Motivation

Despite “dependency style” Generic Haskell, generic functions have a number of restrictions:

▶ only one type argument
▶ no higher-order type-indexed functions
▶ only flat type patterns
▶ complicated types for generic functions of higher arity
▶ no inference of type arguments

Not all of the restrictions pose difficult problems, but all of them are “remaining work”.

Type classes (+extensions) solve many of these problems. Arjan has shown how to encode “dependency style” using type classes.
Getting Rid of Type Classes

Andres Löh

7 July 2004
There are many similarities between type classes and type-indexed functions. But type-indexed functions are better because:

- Type classes create a separate programming language on top of Haskell.
- Type classes seem to have the need of several extensions to acquire their full power.
- Type classes are not first-class either. They are “fixed”.
- Type classes force implicit passing of dictionaries.
Long-term goals

- Extend Haskell language with a type abstraction and type application construct, and a `typedefcase`.
- Type-indexed types take the role of functional dependencies.
- Type system and translation are similar to “dependency style” and type classes: use of qualified types, dictionary passing.
- Type arguments can be inferred in special cases.
- Type arguments can always be specified explicitly.
- Typecases can be open and closed.
- Type-indexed functions are first class.
Long-term goals

- Extend Haskell language with a type abstraction and type application construct, and a **typecase**.
- Type-indexed types take the role of functional dependencies.
- Type system and translation are similar to “dependency style” and type classes: use of qualified types, dictionary passing.
- Type arguments can be inferred in special cases.
- Type arguments can always be specified explicitly.
- Typecases can be open and closed.
- Type-indexed functions are first class.
- Generic functions come (almost) for free.
Long-term goals

- Extend Haskell language with a type abstraction and type application construct, and a **typecase**.
- Type-indexed types take the role of functional dependencies.
- Type system and translation are similar to “dependency style” and type classes: use of qualified types, dictionary passing.
- Type arguments can be inferred in special cases.
- Type arguments can always be specified explicitly.
- Typecases can be open and closed.
- Type-indexed functions are first class.
- Generic functions come (almost) for free.
- This talk: a few small steps.
Pattern Matching for Type-indexed Functions

Andres Löh

7 July 2004
Current situation (Dependency-style)

Patterns are flat.

\[ x \langle T \alpha_1 \ldots \alpha_k \rangle = e \]

Examples:

\[ x \langle [\alpha] \rangle = \ldots \]
\[ x \langle \text{Fix } \varphi \rangle = \ldots \]
\[ x \langle \text{GRose } \varphi \alpha \rangle = \ldots \]

Forbidden:

\[ x \langle [[\text{Int}]] \rangle = \ldots \]
\[ x \langle [[\alpha]] \rangle = \ldots \]
\[ x \langle \text{Either } \alpha \alpha \rangle = \ldots \]
In MPC-style, type patterns are (unapplied) type constructors:

\[
\begin{align*}
  x \langle [] \rangle &= \ldots \\
  x \langle \text{Fix} \rangle &= \ldots \\
  x \langle \text{GRose} \rangle &= \ldots 
\end{align*}
\]

corresponds to

\[
\begin{align*}
  x \langle [\alpha] \rangle &= \ldots \\
  x \langle \text{Fix } \phi \rangle &= \ldots \\
  x \langle \text{GRose } \phi \alpha \rangle &= \ldots 
\end{align*}
\]

in Dependency-style.
Deep patterns are useful

\[
\text{show } \langle [\text{Char}] \rangle x = "\" " \oplus x \oplus "\"
\]
\[
\text{show } \langle [\alpha] \rangle x = "["
\quad + \text{concat } \text{intersperse }","\) \text{(map show } \langle \alpha \rangle x)\) + "]"
\]

\[
\text{flatten } \langle [[\alpha]] \rangle x = \text{flatten } \langle [\alpha] \rangle \text{concat } x
\]
\[
\text{flatten } \langle [\alpha] \rangle x = x
\]
Deep patterns are useful

\[
\text{show } \langle [\text{Char}] \rangle \ x = "\\" \ ++ \ x \ ++ \ "\\"
\]
\[
\text{show } \langle [\alpha] \rangle \ x = "["
\quad + \ \text{concat (intersperse "," (map show } \langle \alpha \rangle \ x))
\quad + \ "]"
\]

\[
\text{flatten } \langle [[[\alpha]]] \rangle \ x = [\text{flatten } \langle [\alpha] \rangle \ \text{concat } x]
\]
\[
\text{flatten } \langle [\alpha] \rangle \ x = x
\]

The order of cases becomes relevant (currently irrelevant):

\[
x \langle (\text{Int}, \alpha) \rangle = 1
\]
\[
x \langle (\alpha, \text{Int}) \rangle = 2
\]
The plan

First, we liberalize the notion of dependencies. Then, we present a translation of a type-indexed function with deep patterns to

- multiple type-indexed functions
- using only flat patterns
- with fallthrough cases (new)
- possibly with multiple type arguments (new)
Liberalized dependencies

Dependencies are currently fixed *per function*. We want to track dependencies *by function case*. Example (from my thesis):

\[
\begin{align*}
\text{equal } & \langle \text{Int} \rangle & = (==) \\
\text{equal } & \langle \text{Unit} \rangle \quad \text{Unit} \quad \text{Unit} & = \text{True} \\
\text{equal } & \langle \text{Sum } \alpha \ \beta \rangle \ (\text{Inl } x) \quad (\text{Inl } y) & = \text{equal } \langle \alpha \rangle \ x \ y \\
\text{equal } & \langle \text{Sum } \alpha \ \beta \rangle \ (\text{Inr } x) \quad (\text{Inr } y) & = \text{equal } \langle \beta \rangle \ x \ y \\
\text{equal } & \langle \text{Sum } \alpha \ \beta \rangle \ _ \quad _ & = \text{False} \\
\text{equal } & \langle \text{Prod } \alpha \ \beta \rangle \ (x_1 \times x_2) \ (y_1 \times y_2) & = \text{equal } \langle \alpha \rangle \ x_1 \ x_2 \ \land \ \text{equal } \langle \beta \rangle \ y_1 \ y_2 \\
\text{equal } & \langle \alpha \rightarrow \beta \rangle \ fx \quad fy & = \text{equal } \langle [\beta] \rangle \ (\text{map } fx \ (\text{enum } \langle \alpha \rangle)) \ (\text{map } fy \ (\text{enum } \langle \alpha \rangle))
\end{align*}
\]

Only one case (for functions) depends on *enum*, but the whole function depends on it.
Currently, this means that a local redefinition for `equal` must redefine `enum` as well:

```haskell
let equal \(\alpha\) x y = toUpper x == toUpper y
enum \(\alpha\) = enum \(Char\)
in equal \([\alpha]\) "laMBdA" "Lambda".
```

- Liberalized dependencies make dependencies variable from case to case.
- In the above redefinition, `enum` would not be needed.
- Only if `equal` is called on function types, `enum` dependencies are passed.
- This is very similar to type classes, which can have different context for different instances.
Liberalized dependencies – contd.

Liberalized dependencies have disadvantages as well:

▷ Type signatures are needed for every case (modulo type inference, which is future work as well).

▷ The qualified type of a function call depends on all dependencies of all cases, whereas now one need only know the type signature of the function.
Nested pattern example: `flatten`

\[
\begin{align*}
\text{flatten } \langle a \rangle & \quad :: \quad (\text{flatten } \langle a \rangle) \Rightarrow a \rightarrow a \\
\text{flatten } \langle [[[\alpha]]] \rangle \; x & = \; [\text{flatten } \langle [[\alpha]] \rangle \; (\text{concat} \; x)] \\
\text{flatten } \langle [\alpha] \rangle & \; \; x = x \\
\end{align*}
\]

Usage:

\[
\begin{align*}
\text{flatten } \langle [[[\text{Int}]]] \rangle \; [[[1,2,3]], [[4,5,6]], [[7,8,9]]] & \Rightarrow [[[1,2,3,4,5,6,7,8,9]]] \\
\end{align*}
\]

A more interesting variant that always returns a list of depth 1 could be written using a type-indexed type.
Example: flatten – contd.

\[
\text{flatten } \langle a \rangle &:: (\text{flatten } \langle a \rangle) \Rightarrow a \rightarrow a \\
\text{flatten } \langle[[\alpha]]\rangle x = [\text{flatten } \langle[\alpha]\rangle (\text{concat } x)] \\
\text{flatten } \langle[\alpha]\rangle x = x
\]

becomes

\[
\text{flatten } \langle a \rangle &:: (\text{flatten } \langle a \rangle, \text{flatten}_1 \langle a \rangle) \Rightarrow a \rightarrow a \\
\text{flatten } \langle[\beta]\rangle & = \text{flatten}_1 \langle\beta\rangle \\
\text{flatten}_1 \langle a \rangle &:: (\text{flatten } \langle a \rangle, \text{flatten}_1 \langle a \rangle) \Rightarrow [a] \rightarrow [a] \\
\text{flatten}_1 \langle[\beta]\rangle x = [\text{flatten } \langle[\beta]\rangle (\text{concat } x)] \\
\text{flatten}_1 \langle\beta\rangle x = x
\]

Note the fallthrough case in \text{flatten}_1.
New concept: Fallthrough cases

- We allow a single dependency variable as a type pattern.
- For a fallthrough case, one component is generated, as for any other case.
- A fallthrough case matches always.
- The translation is similar to the one for generic abstractions.
- In fact, fallthrough cases can be seen as integrating generic abstractions with typecase-based generic definitions.
Fallthrough cases – contd.

\[
\begin{align*}
\text{flatten}_1 \langle a \rangle & \quad :: (\text{flatten} \langle a \rangle, \text{flatten}_1 \langle a \rangle) \Rightarrow [a] \rightarrow [a] \\
\text{flatten}_1 \langle [\beta] \rangle x & = [\text{flatten} \langle [\beta] \rangle (\text{concat} \ x)] \\
\text{flatten}_1 \langle \beta \rangle & \quad x = x
\end{align*}
\]

becomes

\[
\begin{align*}
\text{cp}(\text{flatten}_1, []) & \quad \text{cp}(\text{flatten}, \beta) \text{ cp}(\text{flatten}_1, \beta) \ x = \ldots \\
\text{cp}(\text{flatten}_1, \text{Any}) & \quad \text{cp}(\text{flatten}, \beta) \text{ cp}(\text{flatten}_1, \beta) \ x = \ x
\end{align*}
\]

The call \(\text{flatten}_1 \langle \text{Char} \rangle\) is translated to

\[
\begin{align*}
\text{cp}(\text{flatten}_1, \text{Any}) & \quad \text{cp}(\text{flatten}, \text{Char}) \text{ cp}(\text{flatten}_1, \text{Char})
\end{align*}
\]
Example: flatten – contd.

\[
\text{flatten} \langle[[\text{Int}]]\rangle x
\]
Example: flatten – contd.

\[
\text{flatten } \langle \llbracket \llbracket \text{Int} \rrbracket \rrbracket \rangle x
\]
\[= \{ \text{expansion of type application } \} \]
\[
\text{let } \{\text{flatten } \langle \beta \rangle = \text{flatten } \langle \llbracket \text{Int} \rrbracket \rangle; \text{flatten}_1 \langle \beta \rangle = \text{flatten}_1 \langle \llbracket \text{Int} \rrbracket \rangle \} \]
\[
\text{in } \text{flatten } \langle [\beta] \rangle x
\]
Example: *flatten* – contd.

\[
\text{flatten } \langle[[\text{Int}]]\rangle x \\
= \{\text{expansion of type application}\} \\
\text{let } \{\text{flatten } \langle \beta \rangle = \text{flatten } \langle[[\text{Int}]]\rangle; \text{flatten}_{1} \langle \beta \rangle = \text{flatten}_{1} \langle[[\text{Int}]]\rangle\} \\
\text{in } \text{flatten } \langle[\beta]\rangle x \\
= \{\text{flatten } \langle[\beta]\rangle = \text{flatten}_{1} \langle \beta \rangle\} \\
\text{flatten}_{1} \langle[[\text{Int}]]\rangle x
\]
Example: \textit{flatten} – contd.

\begin{align*}
\text{flatten } \langle[[\text{Int}]] \rangle x \\
&= \{\text{expansion of type application}\} \\
\text{let} \ \{\text{flatten } \langle\beta\rangle = \text{flatten } \langle[[\text{Int}]]\rangle;\text{flatten}_1 \langle\beta\rangle = \text{flatten}_1 \langle[[\text{Int}]]\rangle\} \\
\text{in} \ \text{flatten } \langle[\beta]\rangle x \\
&= \{\text{flatten } \langle[\beta]\rangle = \text{flatten}_1 \langle\beta\rangle\} \\
\text{flatten}_1 \langle[[\text{Int}]]\rangle x \\
&= \{\text{expansion of type application}\} \\
\text{let} \ \text{flatten } \langle\beta\rangle = \text{flatten } \langle[\text{Int}]\rangle \\
\text{flatten}_1 \langle\beta\rangle = \text{flatten}_1 \langle[\text{Int}]\rangle \\
\text{in} \ \text{flatten}_1 \langle[\beta]\rangle x
\end{align*}
Example: $\textit{flatten} – \textit{contd.}$

\[
\textit{flatten} \langle[[\text{Int}]]\rangle x
\]
\[
= \{\text{expansion of type application}\}
\]
\[
\text{let} \{\textit{flatten} \langle\beta\rangle = \textit{flatten} \langle[[\text{Int}]]\rangle; \textit{flatten}_1 \langle\beta\rangle = \textit{flatten}_1 \langle[[\text{Int}]]\rangle\}\n\]
\[
\text{in} \textit{flatten} \langle[[\beta]]\rangle x
\]
\[
= \{\textit{flatten} \langle[[\beta]]\rangle = \textit{flatten}_1 \langle\beta\rangle\}
\]
\[
\textit{flatten}_1 \langle[[\text{Int}]]\rangle x
\]
\[
= \{\text{expansion of type application}\}
\]
\[
\text{let} \textit{flatten} \langle\beta\rangle = \textit{flatten} \langle[[\text{Int}]]\rangle
\]
\[
\textit{flatten}_1 \langle\beta\rangle = \textit{flatten}_1 \langle[[\text{Int}]]\rangle
\]
\[
\text{in} \textit{flatten}_1 \langle[[\beta]]\rangle x
\]
\[
= \{\textit{flatten}_1 \langle[[\beta]]\rangle x = [\textit{flatten} \langle[[\beta]]\rangle (\text{concat} x)]\}
\]
\[
\text{let} \textit{flatten} \langle\beta\rangle = \textit{flatten} \langle[[\text{Int}]]\rangle
\]
\[
\textit{flatten}_1 \langle\beta\rangle = \textit{flatten}_1 \langle[[\text{Int}]]\rangle
\]
\[
\text{in} [\textit{flatten} \langle[[\beta]]\rangle (\text{concat} x)]
Example: *flatten* – contd.

```latex
\textit{flatten} \langle[[[\text{Int}]]]\rangle \ x \\
= \{\text{previous slide}\} \\
\textbf{let} \ \textit{flatten} \ \langle\beta\rangle = \textit{flatten} \ \langle[\text{Int}]\rangle \\
\textit{flatten}_1 \ \langle\beta\rangle = \textit{flatten}_1 \ \langle[\text{Int}]\rangle \\
\textbf{in} \ \left[\textit{flatten} \ \langle[\beta]\rangle \ (\text{concat} \ x)\right]
```
Example: `flatten` – contd.

\[
\text{flatten } \langle[[[\text{Int}]]]\rangle x \\
= \{\text{previous slide}\} \\
\text{let } \text{flatten } \langle\beta\rangle = \text{flatten } \langle[\text{Int}]\rangle \\
\text{flatten}_1 \langle\beta\rangle = \text{flatten}_1 \langle[\text{Int}]\rangle \\
\text{in } [\text{flatten } \langle[\beta]\rangle (\text{concat } x)] \\
= \{\text{flatten } \langle[\beta]\rangle = \text{flatten}_1 \langle\beta\rangle\} \\
[\text{flatten}_1 \langle[\text{Int}]\rangle (\text{concat } x)]
\]
Example: flatten – contd.

\[ \text{flatten} \langle [[[\text{Int}]]] \rangle \ x \]

\[ = \{ \text{previous slide} \} \]

\[ \text{let } \text{flatten} \ (\beta) = \text{flatten} \langle [\text{Int}] \rangle \]

\[ \text{flatten}_1 \ (\beta) = \text{flatten}_1 \langle [\text{Int}] \rangle \]

\[ \text{in } \text{flatten} \langle [\beta] \rangle \ (\text{concat } x) \]

\[ = \{ \text{flatten} \langle [\beta] \rangle \ = \text{flatten}_1 \langle \beta \rangle \} \]

\[ [\text{flatten}_1 \langle [\text{Int}] \rangle \ (\text{concat } x) ] \]

\[ = \{ \text{expansion of type application} \} \]

\[ \text{let } \text{flatten} \ (\beta) = \text{flatten} \langle \text{Int} \rangle \]

\[ \text{flatten}_1 \ (\beta) = \text{flatten}_1 \langle \text{Int} \rangle \]

\[ \text{in } \text{flatten}_1 \langle [\beta] \rangle \ x \]
Example: `flatten` – contd.

\[
\text{flatten } \langle[[[\text{Int}]끼]]\rangle x
\]

\[
= \{\text{previous slide}\} \\
\text{let } \text{flatten } \langle \beta \rangle = \text{flatten } \langle[[\text{Int}]]\rangle \\
\text{flatten}_1 \langle \beta \rangle = \text{flatten}_1 \langle[[\text{Int}]]\rangle \\
\text{in } [\text{flatten } \langle[[\beta]]\rangle (\text{concat } x)]
\]

\[
= \{\text{flatten } \langle[[\beta]]\rangle = \text{flatten}_1 \langle \beta \rangle\} \\
[\text{flatten}_1 \langle[[\text{Int}]]\rangle (\text{concat } x)]
\]

\[
= \{\text{expansion of type application}\} \\
\text{let } \text{flatten } \langle \beta \rangle = \text{flatten } \langle\text{Int}\rangle \\
\text{flatten}_1 \langle \beta \rangle = \text{flatten}_1 \langle\text{Int}\rangle \\
\text{in } \text{flatten}_1 \langle[[\beta]]\rangle x
\]

\[
= \{\text{flatten}_1 \langle[[\beta]]\rangle x = [\text{flatten } \langle[[\beta]]\rangle (\text{concat } x)]\} \\
\text{let } \text{flatten } \langle \beta \rangle = \text{flatten } \langle\text{Int}\rangle \\
\text{flatten}_1 \langle \beta \rangle = \text{flatten}_1 \langle\text{Int}\rangle \\
\text{in } [[[\text{flatten } \langle[[\beta]]\rangle (\text{concat } (\text{concat } x))]]]
Example: flatten – contd.

\[
\text{flatten} \; \langle[[\text{Int}]]\rangle \; x
\]

\[
= \{ \text{previous slide} \}
\]

\[
\text{let flatten} \; \langle \beta \rangle = \text{flatten} \; \langle \text{Int} \rangle
\]

\[
\text{flatten}_1 \; \langle \beta \rangle = \text{flatten}_1 \; \langle \text{Int} \rangle
\]

\[
\text{in} \; \langle[[\text{flatten} \; \langle[[\beta]\rangle \; (\text{concat} \; (\text{concat} \; \text{x}))]]]\rangle
\]
Example: \textit{flatten} – contd.

\[
\text{flatten} \langle[[\text{Int}]]\rangle \ x
\]
\[
= \{\text{previous slide} \} \\
\text{let } \text{flatten} \langle \beta \rangle = \text{flatten} \langle \text{Int} \rangle \\
\text{flatten}_1 \langle \beta \rangle = \text{flatten}_1 \langle \text{Int} \rangle \\
\text{in } \langle \text{flatten} \langle \beta \rangle \ (\text{concat} \ (\text{concat} \ x)) \rangle \rangle
\]
\[
= \{\text{flatten} \langle \beta \rangle = \text{flatten}_1 \langle \beta \rangle \} \\
\langle \text{flatten}_1 \langle \text{Int} \rangle \ (\text{concat} \ (\text{concat} \ x)) \rangle \rangle
\]
Example: \textit{flatten} – contd.

\begin{verbatim}
\texttt{flatten \{\{[\texttt{Int}]\}\}} x \\
= \texttt{\{previous slide \}} \\
\texttt{let flatten \{\texttt{\beta}\} = flatten \{\texttt{Int}\} \\
flatten_1 \{\texttt{\beta}\} = flatten_1 \{\texttt{Int}\} \\
in \{\texttt{flatten \{\[\texttt{\beta}\]\} (concat (concat x))}\}\} \\
= \{\texttt{flatten \{\[\beta\]\} == flatten_1 \{\beta\}\}} \\
\{\texttt{flatten_1 \{\texttt{Int}\} (concat (concat x))}\}\} \\
= \{\texttt{flatten_1 \{\texttt{\beta}\} x == x\} \\
\{\texttt{(concat (concat x))}\}\}
\end{verbatim}
Example: flatten – contd.

The translation of flatten depends on flatten\textsubscript{1}. What happens with local redefinitions?

```ml
let flatten (\alpha\rangle x = reverse x
in flatten (\alpha\rangle [[[1, 2, 3], [4, 5, 6]], [[7, 8, 9]]]
```
Example: \texttt{flatten} – contd.

The translation of \texttt{flatten} depends on \texttt{flatten}_1. What happens with local redefinitions?

\[
\text{let } \texttt{flatten} \; \langle \alpha \rangle \; x = \text{reverse} \; x \\
\text{in } \texttt{flatten} \; \langle [\alpha] \rangle \; [[[1,2,3],[4,5,6]],[[7,8,9]]]
\]

This is translated to:

\[
\text{let } \texttt{flatten} \; \langle \alpha \rangle \; x = \text{reverse} \; x \\
\quad \texttt{flatten}_1 \; \langle \alpha \rangle \; x = x \\
\text{in } \texttt{flatten} \; \langle [\alpha] \rangle \; [[[1,2,3],[4,5,6]],[[7,8,9]]]
\]

The fallthrough case of \texttt{flatten}_1 is added. The result is

\[
[[[7,8,9]],[[1,2,3],[4,5,6]]].
\]
**New concept: Multiple type arguments**

In the general case, we need multiple type arguments.

<table>
<thead>
<tr>
<th></th>
<th>Int, Int</th>
<th>Int, Char</th>
<th>(\langle \alpha, [\text{Int}] \rangle)</th>
<th>Int, (\alpha)</th>
<th>Char</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{poly})</td>
<td>(x, y) = x + y</td>
<td>(x, _) = x</td>
<td>(\langle\alpha, \text{Int} \rangle) (\langle\text{-, } \text{ys}\rangle = \text{maximum } \text{ys})</td>
<td>(x, y) = (x + \text{poly} \langle\alpha\rangle y)</td>
<td>(x) = \text{ord } x</td>
</tr>
</tbody>
</table>
New concept: Multiple type arguments

In the general case, we need multiple type arguments.

\[
\begin{align*}
\text{poly} \langle \text{Int}, \text{Int} \rangle (x, y) &= x + y \\
\text{poly} \langle \text{Int}, \text{Char} \rangle (x, \_ ) &= x \\
\text{poly} \langle \alpha, \text{[Int]} \rangle (\_, ys) &= \text{maximum} \; ys \\
\text{poly} \langle \text{Int}, \alpha \rangle (x, y) &= x + \text{poly} \langle \alpha \rangle y \\
\text{poly} \langle \text{Char} \rangle x &= \text{ord} \; x
\end{align*}
\]

becomes

\[
\begin{align*}
\text{poly} \langle (\alpha, \beta) \rangle &= \text{poly}_1 \langle \alpha \rangle \langle \beta \rangle \\
\text{poly} \langle \text{Char} \rangle x &= \text{ord} \; x \\
\text{poly}_1 \langle \text{Int} \rangle \langle \text{Int} \rangle (x, y) &= x + y \\
\text{poly}_1 \langle \text{Int} \rangle \langle \text{Char} \rangle (x, \_ ) &= x \\
\text{poly}_1 \langle \alpha \rangle \langle [\beta] \rangle &= \text{poly}_2 \langle \alpha \rangle \langle \beta \rangle \\
\text{poly}_1 \langle \alpha \rangle \langle \beta \rangle &= \text{poly}_3 \langle \alpha \rangle \langle \beta \rangle \\
\text{poly}_2 \langle \alpha \rangle \langle \text{Int} \rangle (\_, ys) &= \text{maximum} \; ys \\
\text{poly}_2 \langle \alpha \rangle \langle \beta \rangle &= \text{poly}_3 \langle \alpha \rangle \langle [\beta] \rangle \\
\text{poly}_3 \langle \text{Int} \rangle \langle \alpha \rangle (x, ys) &= x + \text{poly} \langle \alpha \rangle y
\end{align*}
\]
How do multiple type arguments work?

In each case of the definition,
▶ each of the type patterns must be flat,
▶ all type variables of all patterns must be distinct.

When applied,
▶ all type arguments have to be provided.

Furthermore,
▶ Multiple type arguments interact with fallthrough cases.
▶ Multiple type arguments require per-case dependencies.
▶ Multiple type arguments allow to get rid of higher-arity generic functions. For instance, map can be written with two type arguments.
How do multiple type arguments work? In each case of the definition,

- each of the type patterns must be flat,
- all type variables of all patterns must be distinct.
How do multiple type arguments work? In each case of the definition,
  ▶ each of the type patterns must be flat,
  ▶ all type variables of all patterns must be distinct.
When applied,
  ▶ all type arguments have to be provided.
Multiple type arguments – contd.

How do multiple type arguments work?
In each case of the definition,
  ▶ each of the type patterns must be flat,
  ▶ all type variables of all patterns must be distinct.

When applied,
  ▶ all type arguments have to be provided.

Furthermore,
  ▶ Multiple type arguments interact with fallthrough cases.
  ▶ Multiple type arguments require per-case dependencies.
  ▶ Multiple type arguments allow to get rid of higher-arity
generic functions. For instance, map can be written with
two type arguments.
Implementation of multiple type arguments

Once we have liberalized dependencies, they are easy to add.

▶ Each case of the definition is translated to a component.
▶ Components are parametrized by multiple type constructors now.

However:

▶ Specializations are also parametrized by multiple type constructors.
▶ Potential explosion of specializations required, bounded by $d^n$, where $d$ is the number of datatypes and $n$ is the number of type arguments.
▶ In connection with fallthrough cases, code explosion does not occur.
Implementation of multiple type arguments

Once we have liberalized dependencies, they are easy to add.

▶ Each case of the definition is translated to a component.
▶ Components are parametrized by multiple type constructors now.

However:

▶ Specializations are also parametrized by multiple type constructors.
▶ Potential explosion of specializations required, bounded by $d^n$, where $d$ is the number of datatypes and $n$ is the number of type arguments.
▶ In connection with fallthrough cases, code explosion does not occur.
poly_1 \langle \text{Int} \rangle \langle \text{Int} \rangle \ (x, y) = x + y
poly_1 \langle \text{Int} \rangle \langle \text{Char} \rangle \ (x, _) = x
poly_1 \langle \alpha \rangle \langle \beta \rangle = poly_2 \langle \alpha \rangle \langle \beta \rangle
poly_1 \langle \alpha \rangle \langle \beta \rangle = poly_3 \langle \alpha \rangle \langle \beta \rangle

becomes

cp(poly_1, \text{Int} \times \text{Int}) \ (x, y) = x + y
cp(poly_1, \text{Int} \times \text{Char}) \ (x, _) = x
cp(poly_1, \text{Any} \times \text{[]} \rangle \ cp(poly_2, \alpha) (\beta) = cp(poly_2, \alpha) (\beta)
cp(poly_1, \text{Any} \times \text{Any}) \ cp(poly_3, \alpha) (\beta) = cp(poly_3, \alpha) (\beta)

Call translation:

d_\langle \text{Int} \rangle (\text{Int} \times \text{Char}) \simeq cp(poly_1, \text{Any} \times \text{[]}) \ cp(poly_2 \langle \text{Int} \rangle \langle \text{Char} \rangle)
Conclusions

Liberalized dependencies

- make dependency behaviour more similar to type classes
- are necessary to track a large number of dependencies efficiently

Fallthrough cases

- are an important yet simple to implement extension
- are yet another concept next to generic abstraction (allows higher-kinded abstractions) and default cases (allows redirection of dependencies)

Generic functions with multiple type arguments

- are necessary to implement deep patterns
- allow simplification of type system

More to come . . . Comments?
Conclusions

**Liberalized dependencies**
- make dependency behaviour more similar to type classes
- are necessary to track a large number of dependencies efficiently

**Fallthrough cases**
- are an important yet simple to implement extension
- are yet another concept next to generic abstraction (allows higher-kindred abstractions) and default cases (allows redirection of dependencies)
Conclusions

**Liberalized dependencies**

- make dependency behaviour more similar to type classes
- are necessary to track a large number of dependencies efficiently

**Fallthrough cases**

- are an important yet simple to implement extension
- are yet another concept next to generic abstraction (allows higher-kindred abstractions) and default cases (allows redirection of dependencies)

**Generic functions with multiple type arguments**

- are necessary to implement deep patterns
- with liberalized dependencies, allow simplification of type system
Conclusions

Liberalized dependencies
▶ make dependency behaviour more similar to type classes
▶ are necessary to track a large number of dependencies efficiently

Fallthrough cases
▶ are an important yet simple to implement extension
▶ are yet another concept next to generic abstraction (allows higher-kinded abstractions) and default cases (allows redirection of dependencies)

Generic functions with multiple type arguments
▶ are necessary to implement deep patterns
▶ with liberalized dependencies, allow simplification of type system

More to come . . . Comments?
Acknowledgements

Many thanks to Arthur van Leeuwen for taking the time to design this beautiful and (nearly) “huisstijl”-conformant \LaTeX{} theme.