Generalizing Generic Functions

Andres Löh

7 July 2004

Universiteit Utrecht

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

More motivation

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 **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.

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

Universiteit Utrecht

Current situation (Dependency-style)

Patterns are flat.

$$x \ \langle T \ \alpha_1 \ldots \alpha_k \rangle \ = e$$

Examples:

$$\begin{aligned} x \ \langle [\alpha] \rangle \quad &= \ \ldots \\ x \ \langle \mathit{Fix} \ \varphi \rangle \quad &= \ \ldots \\ x \ \langle \mathit{GRose} \ \varphi \ \alpha \rangle \quad &= \ \ldots \end{aligned}$$

Forbidden:

$$\begin{aligned} x \ \langle [\mathit{Int}] \rangle \quad &= \ \ldots \\ x \ \langle [[\alpha]] \rangle \quad &= \ \ldots \\ x \ \langle \mathit{Either} \ \alpha \ \alpha \rangle \quad &= \ \ldots \end{aligned}$$

Historical reasons (MPC-style)

In MPC-style, type patterns are (unapplied) type constructors:

$$
\begin{aligned}
&x\ \langle [\,]\rangle &&= \ldots\\
&x\ \langle \mathit{Fix}\rangle &&= \ldots\\
&x\ \langle \mathit{GRose}\rangle &&= \ldots
\end{aligned}
$$

corresponds to

$$
\begin{aligned}
&x\ \langle [\alpha]\rangle &&= \ldots\\
&x\ \langle \mathit{Fix}\ \varphi\rangle &&= \ldots\\
&x\ \langle \mathit{GRose}\ \varphi\ \alpha\rangle &&= \ldots
\end{aligned}
$$

in Dependency-style.

Deep patterns are useful

```
show ⟨[Char]⟩ x = "\"" ++ x ++ "\""
show ⟨[α]⟩    x =    "["
                  ++ concat (intersperse "," (map show ⟨α⟩ x))
                  ++ "]"
```

```
flatten ⟨[[α]]⟩ x = [flatten ⟨[α]⟩ concat x]
flatten ⟨[α]⟩   x = x
```

Deep patterns are useful

$$
\begin{aligned}
&\mathit{show}\ \langle[\mathit{Char}]\rangle\ x = \texttt{"\textbackslash""} +\!\!+\ x +\!\!+ \texttt{"\textbackslash""}\\
&\mathit{show}\ \langle[\alpha]\rangle\ \quad x = \quad \texttt{"["}\\
&\qquad\qquad\qquad +\!\!+\ \mathit{concat}\ (\mathit{intersperse}\ \texttt{","}\ (\mathit{map}\ \mathit{show}\ \langle\alpha\rangle\ x))\\
&\qquad\qquad\qquad +\!\!+\ \texttt{"]"}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{flatten}\ \langle[[\alpha]]\rangle\ x = [\mathit{flatten}\ \langle[\alpha]\rangle\ \mathit{concat}\ x]\\
&\mathit{flatten}\ \langle[\alpha]\rangle\ \quad x = x
\end{aligned}
$$

The order of cases becomes relevant (currently irrelevant):

$$
\begin{aligned}
&x\ \langle(\mathit{Int},\alpha)\rangle \quad = 1\\
&x\ \langle(\alpha,\mathit{Int})\rangle \quad = 2
\end{aligned}
$$

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):

$$
\begin{array}{lllll}
\mathit{equal}\ \langle \mathit{Int} \rangle & & & = & (==) \\
\mathit{equal}\ \langle \mathit{Unit} \rangle & \mathit{Unit} & \mathit{Unit} & = & \mathit{True} \\
\mathit{equal}\ \langle \mathit{Sum}\ \alpha\ \beta \rangle & (\mathit{Inl}\ x) & (\mathit{Inl}\ y) & = & \mathit{equal}\ \langle \alpha \rangle\ x\ y \\
\mathit{equal}\ \langle \mathit{Sum}\ \alpha\ \beta \rangle & (\mathit{Inr}\ x) & (\mathit{Inr}\ y) & = & \mathit{equal}\ \langle \beta \rangle\ x\ y \\
\mathit{equal}\ \langle \mathit{Sum}\ \alpha\ \beta \rangle & \_ & \_ & = & \mathit{False} \\
\mathit{equal}\ \langle \mathit{Prod}\ \alpha\ \beta \rangle & (x_1 \times x_2) & (y_1 \times y_2) & = & \mathit{equal}\ \langle \alpha \rangle\ x_1\ x_2 \wedge \mathit{equal}\ \langle \beta \rangle\ y_1\ y_2 \\
\mathit{equal}\ \langle \alpha \to \beta \rangle & \mathit{fx} & \mathit{fy} & = & \mathit{equal}\ \langle [\beta] \rangle\ \big(\mathit{map}\ \mathit{fx}\ (\mathit{enum}\ \langle \alpha \rangle)\big) \\
 & & & & \qquad\qquad\quad \big(\mathit{map}\ \mathit{fy}\ (\mathit{enum}\ \langle \alpha \rangle)\big)
\end{array}
$$

Only one case (for functions) depends on *enum*, but the whole function depends on it.

Liberalized dependencies – contd.

Currently, this means that a local redefinition for *equal* must redefine *enum* as well:

$$\begin{array}{l} \textbf{let}\ equal\ \langle\alpha\rangle\ x\ y = toUpper\ x == toUpper\ y \\ \quad\ enum\ \langle\alpha\rangle \qquad = enum\ \langle Char\rangle \\ \textbf{in}\ \ equal\ \langle[\alpha]\rangle\ \texttt{"laMBdA"}\ \texttt{"Lambda"}\ . \end{array}$$

- 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*

$$
\begin{aligned}
&\mathit{flatten}\ \langle a\rangle &&:: (\mathit{flatten}\ \langle a\rangle) \Rightarrow a \to a\\
&\mathit{flatten}\ \langle[[\alpha]]\rangle\ x &&= [\mathit{flatten}\ \langle[\alpha]\rangle\ (\mathit{concat}\ x)]\\
&\mathit{flatten}\ \langle[\alpha]\rangle\ \ x &&= x
\end{aligned}
$$

Usage:

$$
\begin{aligned}
&\mathit{flatten}\ \langle[[[\mathit{Int}]]]\rangle\ [[[1,2,3],[4,5,6]],[[7,8,9]]]\\
&\qquad\qquad \leadsto [[[1,2,3,4,5,6,7,8,9]]]
\end{aligned}
$$

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

Example: *flatten* – contd.

$$\begin{aligned}
&\mathit{flatten}\ \langle a\rangle && :: (\mathit{flatten}\ \langle a\rangle) \Rightarrow a \to a\\
&\mathit{flatten}\ \langle[[\alpha]]\rangle\ x &&= [\mathit{flatten}\ \langle[\alpha]\rangle\ (\mathit{concat}\ x)]\\
&\mathit{flatten}\ \langle[\alpha]\rangle\ x &&= x
\end{aligned}$$

becomes

$$\begin{aligned}
&\mathit{flatten}\ \langle a\rangle && :: (\mathit{flatten}\ \langle a\rangle, \mathit{flatten}_1\ \langle a\rangle) \Rightarrow a \to a\\
&\mathit{flatten}\ \langle[\beta]\rangle &&= \mathit{flatten}_1\ \langle\beta\rangle\\
&\mathit{flatten}_1\ \langle a\rangle && :: (\mathit{flatten}\ \langle a\rangle, \mathit{flatten}_1\ \langle a\rangle) \Rightarrow [a] \to [a]\\
&\mathit{flatten}_1\ \langle[\beta]\rangle\ x &&= [\mathit{flatten}\ \langle[\beta]\rangle\ (\mathit{concat}\ x)]\\
&\mathit{flatten}_1\ \langle\beta\rangle\ x &&= x
\end{aligned}$$

Note the fallthrough case in $\mathit{flatten}_1$.

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.

$$\begin{aligned}
&\mathit{flatten}_1\ \langle a\rangle && :: (\mathit{flatten}\ \langle a\rangle, \mathit{flatten}_1\ \langle a\rangle) \Rightarrow [a] \to [a] \\
&\mathit{flatten}_1\ \langle [\beta]\rangle\ x && = [\mathit{flatten}\ \langle [\beta]\rangle\ (\mathit{concat}\ x)] \\
&\mathit{flatten}_1\ \langle \beta\rangle\ \ \ x && = x
\end{aligned}$$

becomes

$$\begin{aligned}
&\mathsf{cp}(\mathit{flatten}_1, [\,])\ \ \ \mathsf{cp}(\mathit{flatten}, \beta)\ \mathsf{cp}(\mathit{flatten}_1, \beta)\ x = \ \ldots \\
&\mathsf{cp}(\mathit{flatten}_1, \mathsf{Any})\ \mathsf{cp}(\mathit{flatten}, \beta)\ \mathsf{cp}(\mathit{flatten}_1, \beta)\ x = \ \ x
\end{aligned}$$

The call $\mathit{flatten}_1\ \langle \mathit{Char}\rangle$ is translated to

$$\mathsf{cp}(\mathit{flatten}_1, \mathsf{Any})\ \mathsf{cp}(\mathit{flatten}, \mathit{Char})\ \mathsf{cp}(\mathit{flatten}_1, \mathit{Char})$$

Example: *flatten* – contd.

- $flatten\ \langle[[[Int]]]\rangle\ x$

Example: *flatten* – contd.

$$
\begin{array}{ll}
& \mathit{flatten}\ \langle[[[\mathit{Int}]]]\rangle\ x \\
== & \{\text{expansion of type application}\ \} \\
& \textbf{let}\ \{\mathit{flatten}\ \langle\beta\rangle = \mathit{flatten}\ \langle[[\mathit{Int}]]\rangle; \mathit{flatten}_1\ \langle\beta\rangle = \mathit{flatten}_1\ \langle[[\mathit{Int}]]\rangle\} \\
& \textbf{in}\ \ \mathit{flatten}\ \ \langle[\beta]\rangle\ x
\end{array}
$$

Example: *flatten* – contd.

$$flatten\ \langle[[[Int]]]\rangle\ x$$

$$\doteq \{\text{expansion of type application}\}$$

$$\textbf{let}\ \{flatten\ \langle\beta\rangle = flatten\ \langle[[Int]]\rangle; flatten_1\ \langle\beta\rangle = flatten_1\ \langle[[Int]]\rangle\}$$
$$\textbf{in}\ flatten\ \langle[\beta]\rangle\ x$$

$$\doteq \{flatten\ \langle[\beta]\rangle \doteq flatten_1\ \langle\beta\rangle\}$$

$$flatten_1\ \langle[[Int]]\rangle\ x$$

Example: *flatten* – contd.

$$
\begin{aligned}
& \mathit{flatten}\ \langle[[[\mathit{Int}]]]\rangle\ x \\
\equiv\ & \{\text{expansion of type application}\} \\
& \textbf{let}\ \{\mathit{flatten}\ \langle\beta\rangle = \mathit{flatten}\ \langle[[\mathit{Int}]]\rangle; \mathit{flatten}_1\ \langle\beta\rangle = \mathit{flatten}_1\ \langle[[\mathit{Int}]]\rangle\} \\
& \textbf{in}\ \ \mathit{flatten}\ \ \langle[\beta]\rangle\ x \\
\equiv\ & \{\mathit{flatten}\ \langle[\beta]\rangle \equiv \mathit{flatten}_1\ \langle\beta\rangle\} \\
& \mathit{flatten}_1\ \langle[[\mathit{Int}]]\rangle\ x \\
\equiv\ & \{\text{expansion of type application}\} \\
& \textbf{let}\ \mathit{flatten}\ \ \langle\beta\rangle = \mathit{flatten}\ \ \langle[\mathit{Int}]\rangle \\
& \qquad \mathit{flatten}_1\ \langle\beta\rangle = \mathit{flatten}_1\ \langle[\mathit{Int}]\rangle \\
& \textbf{in}\ \ \mathit{flatten}_1\ \langle[\beta]\rangle\ x
\end{aligned}
$$

Example: *flatten* – contd.

$$
\begin{array}{l}
\quad \mathit{flatten}\ \langle[[[\mathit{Int}]]]\rangle\ x \\
\equiv \{\text{expansion of type application}\} \\
\quad \textbf{let}\ \{\mathit{flatten}\ \langle\beta\rangle = \mathit{flatten}\ \langle[[\mathit{Int}]]\rangle; \mathit{flatten}_1\ \langle\beta\rangle = \mathit{flatten}_1\ \langle[[\mathit{Int}]]\rangle\} \\
\quad \textbf{in}\ \mathit{flatten}\ \langle[\beta]\rangle\ x \\
\equiv \{\mathit{flatten}\ \langle[\beta]\rangle \equiv \mathit{flatten}_1\ \langle\beta\rangle\} \\
\quad \mathit{flatten}_1\ \langle[[\mathit{Int}]]\rangle\ x \\
\equiv \{\text{expansion of type application}\} \\
\quad \textbf{let}\ \mathit{flatten}\ \langle\beta\rangle = \mathit{flatten}\ \langle[\mathit{Int}]\rangle \\
\qquad\ \ \mathit{flatten}_1\ \langle\beta\rangle = \mathit{flatten}_1\ \langle[\mathit{Int}]\rangle \\
\quad \textbf{in}\ \mathit{flatten}_1\ \langle[\beta]\rangle\ x \\
\equiv \{\mathit{flatten}_1\ \langle[\beta]\rangle\ x \equiv [\mathit{flatten}\ \langle[\beta]\rangle\ (\mathit{concat}\ x)]\} \\
\quad \textbf{let}\ \mathit{flatten}\ \langle\beta\rangle = \mathit{flatten}\ \langle[\mathit{Int}]\rangle \\
\qquad\ \ \mathit{flatten}_1\ \langle\beta\rangle = \mathit{flatten}_1\ \langle[\mathit{Int}]\rangle \\
\quad \textbf{in}\ [\mathit{flatten}\ \langle[\beta]\rangle\ (\mathit{concat}\ x)]
\end{array}
$$

Example: *flatten* – contd.

$$
\begin{aligned}
& \quad \mathit{flatten}\ \langle[[[\mathit{Int}]]]\rangle\ x \\
\equiv\ & \{\text{previous slide}\} \\
& \quad \textbf{let}\ \mathit{flatten}\ \ \langle\beta\rangle = \mathit{flatten}\ \ \langle[\mathit{Int}]\rangle \\
& \quad \phantom{\textbf{let}}\ \mathit{flatten}_1\ \langle\beta\rangle = \mathit{flatten}_1\ \langle[\mathit{Int}]\rangle \\
& \quad \textbf{in}\ \ [\mathit{flatten}\ \langle[\beta]\rangle\ (\mathit{concat}\ x)]
\end{aligned}
$$

Example: *flatten* – contd.

$$
\begin{array}{ll}
 & \mathit{flatten}\ \langle[[[\mathit{Int}]]]\rangle\ x \\
\equiv & \{\text{previous slide}\} \\
 & \textbf{let}\ \mathit{flatten}\ \ \langle\beta\rangle = \mathit{flatten}\ \ \langle[\mathit{Int}]\rangle \\
 & \qquad \mathit{flatten}_1\ \langle\beta\rangle = \mathit{flatten}_1\ \langle[\mathit{Int}]\rangle \\
 & \textbf{in}\ \ [\mathit{flatten}\ \langle[\beta]\rangle\ (\mathit{concat}\ x)] \\
\equiv & \{\mathit{flatten}\ \langle[\beta]\rangle \equiv \mathit{flatten}_1\ \langle\beta\rangle\} \\
 & [\mathit{flatten}_1\ \langle[\mathit{Int}]\rangle\ (\mathit{concat}\ x)]
\end{array}
$$

Example: *flatten* – contd.

$$
\begin{array}{ll}
 & \mathit{flatten}\ \langle[[[\mathit{Int}]]]\rangle\ x \\
\equiv & \{\text{previous slide}\} \\
 & \textbf{let}\ \mathit{flatten}\ \langle\beta\rangle = \mathit{flatten}\ \langle[\mathit{Int}]\rangle \\
 & \quad\ \ \mathit{flatten}_1\ \langle\beta\rangle = \mathit{flatten}_1\ \langle[\mathit{Int}]\rangle \\
 & \textbf{in}\ [\mathit{flatten}\ \langle[\beta]\rangle\ (\mathit{concat}\ x)] \\
\equiv & \{\mathit{flatten}\ \langle[\beta]\rangle \equiv \mathit{flatten}_1\ \langle\beta\rangle\} \\
 & [\mathit{flatten}_1\ \langle[\mathit{Int}]\rangle\ (\mathit{concat}\ x)] \\
\equiv & \{\text{expansion of type application}\} \\
 & \textbf{let}\ \mathit{flatten}\ \langle\beta\rangle = \mathit{flatten}\ \langle\mathit{Int}\rangle \\
 & \quad\ \ \mathit{flatten}_1\ \langle\beta\rangle = \mathit{flatten}_1\ \langle\mathit{Int}\rangle \\
 & \textbf{in}\ \mathit{flatten}_1\ \langle[\beta]\rangle\ x
\end{array}
$$

Example: *flatten* – contd.

$$
\begin{aligned}
& \mathit{flatten}\ \langle[[[Int]]]\rangle\ x \\
== & \ \{\text{previous slide}\} \\
& \textbf{let}\ \mathit{flatten}\ \langle\beta\rangle = \mathit{flatten}\ \langle[Int]\rangle \\
& \qquad \mathit{flatten}_1\ \langle\beta\rangle = \mathit{flatten}_1\ \langle[Int]\rangle \\
& \textbf{in}\ [\mathit{flatten}\ \langle[\beta]\rangle\ (\mathit{concat}\ x)] \\
== & \ \{\mathit{flatten}\ \langle[\beta]\rangle == \mathit{flatten}_1\ \langle\beta\rangle\} \\
& [\mathit{flatten}_1\ \langle[Int]\rangle\ (\mathit{concat}\ x)] \\
== & \ \{\text{expansion of type application}\} \\
& \textbf{let}\ \mathit{flatten}\ \langle\beta\rangle = \mathit{flatten}\ \langle Int\rangle \\
& \qquad \mathit{flatten}_1\ \langle\beta\rangle = \mathit{flatten}_1\ \langle Int\rangle \\
& \textbf{in}\ \mathit{flatten}_1\ \langle[\beta]\rangle\ x \\
== & \ \{\mathit{flatten}_1\ \langle[\beta]\rangle\ x == [\mathit{flatten}\ \langle[\beta]\rangle\ (\mathit{concat}\ x)]\} \\
& \textbf{let}\ \mathit{flatten}\ \langle\beta\rangle = \mathit{flatten}\ \langle Int\rangle \\
& \qquad \mathit{flatten}_1\ \langle\beta\rangle = \mathit{flatten}_1\ \langle Int\rangle \\
& \textbf{in}\ [[\mathit{flatten}\ \langle[\beta]\rangle\ (\mathit{concat}\ (\mathit{concat}\ x))]]
\end{aligned}
$$

Example: *flatten* – contd.

$$
\begin{array}{l}
\quad \mathit{flatten}\ \langle[[[\mathit{Int}]]]\rangle\ x \\
== \{\text{previous slide}\} \\
\quad \textbf{let}\ \mathit{flatten}\ \ \langle\beta\rangle = \mathit{flatten}\ \ \langle\mathit{Int}\rangle \\
\quad \phantom{\textbf{let}}\ \mathit{flatten}_1\ \langle\beta\rangle = \mathit{flatten}_1\ \langle\mathit{Int}\rangle \\
\quad \textbf{in}\ \ [[\mathit{flatten}\ \langle[\beta]\rangle\ (\mathit{concat}\ (\mathit{concat}\ x))]]
\end{array}
$$

Example: *flatten* – contd.

$$
\begin{array}{ll}
& \mathit{flatten}\ \langle[[[\mathit{Int}]]]\rangle\ x \\
\equiv & \{\text{previous slide}\} \\
& \mathbf{let}\ \mathit{flatten}\ \ \langle\beta\rangle = \mathit{flatten}\ \ \langle\mathit{Int}\rangle \\
& \qquad \mathit{flatten}_1\ \langle\beta\rangle = \mathit{flatten}_1\ \langle\mathit{Int}\rangle \\
& \mathbf{in}\ \ [\![\mathit{flatten}\ \langle[\beta]\rangle\ (\mathit{concat}\ (\mathit{concat}\ x))]\!] \\
\equiv & \{\mathit{flatten}\ \langle[\beta]\rangle \equiv \mathit{flatten}_1\ \langle\beta\rangle\} \\
& [\![\mathit{flatten}_1\ \langle\mathit{Int}\rangle\ (\mathit{concat}\ (\mathit{concat}\ x))]\!]
\end{array}
$$

Example: *flatten* – contd.

$$
\begin{array}{l}
\quad \mathit{flatten}\ \langle[[[\mathit{Int}]]]\rangle\ x \\
== \{\text{previous slide}\} \\
\quad \mathbf{let}\ \mathit{flatten}\ \ \langle\beta\rangle = \mathit{flatten}\ \ \langle \mathit{Int}\rangle \\
\qquad\ \ \mathit{flatten}_1\ \langle\beta\rangle = \mathit{flatten}_1\ \langle \mathit{Int}\rangle \\
\quad \mathbf{in}\ \ [[\mathit{flatten}\ \langle[\beta]\rangle\ (\mathit{concat}\ (\mathit{concat}\ x))]] \\
== \{\mathit{flatten}\ \langle[\beta]\rangle == \mathit{flatten}_1\ \langle\beta\rangle\} \\
\quad [[\mathit{flatten}_1\ \langle \mathit{Int}\rangle\ (\mathit{concat}\ (\mathit{concat}\ x))]] \\
== \{\mathit{flatten}_1\ \langle\beta\rangle\ x == x\} \\
\quad [[(\mathit{concat}\ (\mathit{concat}\ x))]]
\end{array}
$$

Example: *flatten* – contd.

The translation of *flatten* depends on $flatten_1$. What happens with local redefinitions?

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

Example: *flatten* – contd.

The translation of *flatten* depends on $flatten_1$. What happens with local redefinitions?

$$\begin{array}{l}\textbf{let}\ \mathit{flatten}\ \langle\alpha\rangle\ x = \mathit{reverse}\ x \\ \textbf{in}\ \ \mathit{flatten}\ \langle[\alpha]\rangle\ [[[1,2,3],[4,5,6]],[[7,8,9]]]\end{array}$$

This is translated to:

$$\begin{array}{l}\textbf{let}\ \mathit{flatten}\ \ \langle\alpha\rangle\ x = \mathit{reverse}\ x \\ \qquad \mathit{flatten}_1\ \langle\alpha\rangle\ x = x \\ \textbf{in}\ \ \mathit{flatten}\ \ \langle[\alpha]\rangle\ [[[1,2,3],[4,5,6]],[[7,8,9]]]\end{array}$$

The fallthrough case of $flatten_1$ 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.

$$
\begin{array}{lll}
\mathit{poly}\ \langle \mathit{Int}, \mathit{Int} \rangle & (x, y) & = x + y \\
\mathit{poly}\ \langle \mathit{Int}, \mathit{Char} \rangle & (x, \_) & = x \\
\mathit{poly}\ (\langle \alpha, [\mathit{Int}] \rangle & (\_, ys) & = \mathit{maximum}\ ys \\
\mathit{poly}\ \langle \mathit{Int}, \alpha \rangle & (x, y) & = x + \mathit{poly}\ \langle \alpha \rangle\ y \\
\mathit{poly}\ \langle \mathit{Char} \rangle & x & = \mathit{ord}\ x
\end{array}
$$

New concept: Multiple type arguments

In the general case, we need multiple type arguments.

$$
\begin{array}{llll}
\mathit{poly} & \langle \mathit{Int}, \mathit{Int} \rangle & (x, y) & = x + y \\
\mathit{poly} & \langle \mathit{Int}, \mathit{Char} \rangle & (x, \_) & = x \\
\mathit{poly} & (\langle \alpha, [\mathit{Int}] \rangle & (\_, \mathit{ys}) & = \mathit{maximum}\ \mathit{ys} \\
\mathit{poly} & \langle \mathit{Int}, \alpha \rangle & (x, y) & = x + \mathit{poly}\ \langle \alpha \rangle\ y \\
\mathit{poly} & \langle \mathit{Char} \rangle & x & = \mathit{ord}\ x
\end{array}
$$

becomes

$$
\begin{array}{lllll}
\mathit{poly} & \langle (\alpha, \beta) \rangle & & & = \mathit{poly}_1\ \langle \alpha \rangle\ \langle \beta \rangle \\
\mathit{poly} & \langle \mathit{Char} \rangle & & x & = \mathit{ord}\ x \\
\mathit{poly}_1 & \langle \mathit{Int} \rangle & \langle \mathit{Int} \rangle & (x, y) & = x + y \\
\mathit{poly}_1 & \langle \mathit{Int} \rangle & \langle \mathit{Char} \rangle & (x, \_) & = x \\
\mathit{poly}_1 & \langle \alpha \rangle & \langle [\beta] \rangle & & = \mathit{poly}_2\ \langle \alpha \rangle\ \langle \beta \rangle \\
\mathit{poly}_1 & \langle \alpha \rangle & \langle \beta \rangle & & = \mathit{poly}_3\ \langle \alpha \rangle\ \langle \beta \rangle \\
\mathit{poly}_2 & \langle \alpha \rangle & \langle \mathit{Int} \rangle & (\_, \mathit{ys}) & = \mathit{maximum}\ \mathit{ys} \\
\mathit{poly}_2 & \langle \alpha \rangle & \langle \beta \rangle & & = \mathit{poly}_3\ \langle \alpha \rangle\ \langle [\beta] \rangle \\
\mathit{poly}_3 & \langle \mathit{Int} \rangle & \langle \alpha \rangle & (x, \mathit{ys}) & = x + \mathit{poly}\ \langle \alpha \rangle\ y
\end{array}
$$

Multiple type arguments – contd.

How do multiple type arguments work?

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.

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.

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.

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 – contd.

$$\begin{aligned}
&poly_1\ \langle Int\rangle\ \langle Int\rangle && (x,y) && = x+y\\
&poly_1\ \langle Int\rangle\ \langle Char\rangle && (x,_) && = x\\
&poly_1\ \langle \alpha\rangle\ \langle [\beta]\rangle && && = poly_2\ \langle \alpha\rangle\ \langle \beta\rangle\\
&poly_1\ \langle \alpha\rangle\ \langle \beta\rangle && && = poly_3\ \langle \alpha\rangle\ \langle \beta\rangle
\end{aligned}$$

becomes

$$\begin{aligned}
&\mathsf{cp}(poly_1, Int \times Int) && (x,y) && = x+y\\
&\mathsf{cp}(poly_1, Int \times Char) && (x,_) && = x\\
&\mathsf{cp}(poly_1, Any \times [\,]) && \mathsf{cp}(poly_2,\alpha)\ (\beta) && = \mathsf{cp}(poly_2,\alpha)\ (\beta)\\
&\mathsf{cp}(poly_1, Any \times Any) && \mathsf{cp}(poly_3,\alpha)\ (\beta) && = \mathsf{cp}(poly_3,\alpha)\ (\beta)
\end{aligned}$$

Call translation:

$$poly_1\ \langle Int\rangle\ \langle [Char]\rangle \leadsto \mathsf{cp}(poly_1, Any \times [\,])\ (poly_2\ \langle Int\rangle\ \langle Char\rangle)$$

Conclusions

Liberalized dependencies

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

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)

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

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.