Dependencies — by case or by function?

Andres Löh

5 September 2004

Universiteit Utrecht

Generic equality in Dependency-style

Generic functions are defined by "pattern matching" on a type argument:

$$
\begin{array}{llll}
\mathit{equal}\ \langle \mathrm{Int} \rangle & & & = (\texttt{==}) \\
\mathit{equal}\ \langle \mathrm{Unit} \rangle & \mathit{Unit} & \mathit{Unit} & = \mathit{True} \\
\mathit{equal}\ \langle \mathrm{Sum}\ \alpha\ \beta \rangle & (\mathit{Inl}\ x) & (\mathit{Inl}\ y) & = \mathit{equal}\ \langle \alpha \rangle\ x\ y \\
\mathit{equal}\ \langle \mathrm{Sum}\ \alpha\ \beta \rangle & (\mathit{Inr}\ x) & (\mathit{Inr}\ y) & = \mathit{equal}\ \langle \beta \rangle\ x\ y \\
\mathit{equal}\ \langle \mathrm{Sum}\ \alpha\ \beta \rangle & \_ & \_ & = \mathit{False} \\
\mathit{equal}\ \langle \mathrm{Prod}\ \alpha\ \beta \rangle & (x_1 \times x_2) & (y_1 \times y_2) & = \quad \mathit{equal}\ \langle \alpha \rangle\ x_1\ y_1 \\
 & & & \quad\ \wedge\ \mathit{equal}\ \langle \beta \rangle\ x_2\ y_2\ .
\end{array}
$$

Generic equality in Dependency-style

Generic functions are defined by "pattern matching" on a type argument:

$$
\begin{aligned}
&\mathit{equal}\ \langle \mathrm{Int} \rangle && && &&= (==)\\
&\mathit{equal}\ \langle \mathrm{Unit} \rangle && \mathit{Unit} && \mathit{Unit} &&= \mathit{True}\\
&\mathit{equal}\ \langle \mathrm{Sum}\ \alpha\ \beta \rangle && (\mathit{Inl}\ x) && (\mathit{Inl}\ y) &&= \mathit{equal}\ \langle \alpha \rangle\ x\ y\\
&\mathit{equal}\ \langle \mathrm{Sum}\ \alpha\ \beta \rangle && (\mathit{Inr}\ x) && (\mathit{Inr}\ y) &&= \mathit{equal}\ \langle \beta \rangle\ x\ y\\
&\mathit{equal}\ \langle \mathrm{Sum}\ \alpha\ \beta \rangle && \_ && \_ &&= \mathit{False}\\
&\mathit{equal}\ \langle \mathrm{Prod}\ \alpha\ \beta \rangle && (x_1 \times x_2) && (y_1 \times y_2) &&= \quad \mathit{equal}\ \langle \alpha \rangle\ x_1\ y_1\\
& && && && \quad \wedge\ \mathit{equal}\ \langle \beta \rangle\ x_2\ y_2\ .
\end{aligned}
$$

The types Unit, Sum and Prod are used to represent the structure of Haskell datatypes.

$$
\begin{aligned}
&\mathbf{data}\ \mathrm{Unit} &&= \mathit{Unit}\\
&\mathbf{data}\ \mathrm{Sum}\ a\ b &&= \mathit{Inl}\ a \mid \mathit{Inr}\ b\\
&\mathbf{data}\ \mathrm{Prod}\ a\ b &&= a \times b
\end{aligned}
$$

Generic equality in Dependency-style

Generic functions are defined by "pattern matching" on a type argument:

$$
\begin{array}{llll}
\mathit{equal}\ \langle \mathrm{Int} \rangle & & & = (==) \\
\mathit{equal}\ \langle \mathrm{Unit} \rangle & \mathit{Unit} & \mathit{Unit} & = \mathit{True} \\
\mathit{equal}\ \langle \mathrm{Sum}\ \alpha\ \beta \rangle & (\mathit{Inl}\ x) & (\mathit{Inl}\ y) & = \mathit{equal}\ \langle \alpha \rangle\ x\ y \\
\mathit{equal}\ \langle \mathrm{Sum}\ \alpha\ \beta \rangle & (\mathit{Inr}\ x) & (\mathit{Inr}\ y) & = \mathit{equal}\ \langle \beta \rangle\ x\ y \\
\mathit{equal}\ \langle \mathrm{Sum}\ \alpha\ \beta \rangle & \_ & \_ & = \mathit{False} \\
\mathit{equal}\ \langle \mathrm{Prod}\ \alpha\ \beta \rangle & (x_1 \times x_2) & (y_1 \times y_2) & = \quad \mathit{equal}\ \langle \alpha \rangle\ x_1\ y_1 \\
 & & & \quad \wedge\ \mathit{equal}\ \langle \beta \rangle\ x_2\ y_2\ .
\end{array}
$$

Haskell datatypes are mapped to these datatypes:

$$
\begin{array}{ll}
\mathbf{data}\ \mathrm{List}\ a & = \mathit{Nil} \mid \mathit{Cons}\ a\ (\mathrm{List}\ a) \\
\mathbf{type}\ \mathrm{STR(List)}\ a & = \mathrm{Sum\ Unit}\ (\mathrm{Prod}\ a\ (\mathrm{List}\ a))\ .
\end{array}
$$

Generic equality in Dependency-style

Generic functions are defined by "pattern matching" on a type argument:

$$
\begin{array}{llll}
\mathit{equal}\ \langle \mathrm{Int} \rangle & & & = (==) \\
\mathit{equal}\ \langle \mathrm{Unit} \rangle & \mathit{Unit} & \mathit{Unit} & = \mathit{True} \\
\mathit{equal}\ \langle \mathrm{Sum}\ \alpha\ \beta \rangle & (\mathit{Inl}\ x) & (\mathit{Inl}\ y) & = \mathit{equal}\ \langle \alpha \rangle\ x\ y \\
\mathit{equal}\ \langle \mathrm{Sum}\ \alpha\ \beta \rangle & (\mathit{Inr}\ x) & (\mathit{Inr}\ y) & = \mathit{equal}\ \langle \beta \rangle\ x\ y \\
\mathit{equal}\ \langle \mathrm{Sum}\ \alpha\ \beta \rangle & \_ & \_ & = \mathit{False} \\
\mathit{equal}\ \langle \mathrm{Prod}\ \alpha\ \beta \rangle & (x_1 \times x_2) & (y_1 \times y_2) & = \quad \mathit{equal}\ \langle \alpha \rangle\ x_1\ y_1 \\
 & & & \quad \wedge\ \mathit{equal}\ \langle \beta \rangle\ x_2\ y_2\ .
\end{array}
$$

With above definition,

$$\mathit{equal}\ \langle [\mathrm{Bool}] \rangle\ [\mathit{True}, \mathit{False}]\ [\mathit{True}, \mathit{False}]$$

compiles and evaluates to *True*.

Generic equality in Dependency-style

Generic functions are defined by "pattern matching" on a type argument:

$$
\begin{array}{lllll}
\mathit{equal}\ \langle \mathrm{Int} \rangle & & & = & (==) \\
\mathit{equal}\ \langle \mathrm{Unit} \rangle & \mathit{Unit} & \mathit{Unit} & = & \mathit{True} \\
\mathit{equal}\ \langle \mathrm{Sum}\ \alpha\ \beta \rangle & (\mathit{Inl}\ x) & (\mathit{Inl}\ y) & = & \mathit{equal}\ \langle \alpha \rangle\ x\ y \\
\mathit{equal}\ \langle \mathrm{Sum}\ \alpha\ \beta \rangle & (\mathit{Inr}\ x) & (\mathit{Inr}\ y) & = & \mathit{equal}\ \langle \beta \rangle\ x\ y \\
\mathit{equal}\ \langle \mathrm{Sum}\ \alpha\ \beta \rangle & \_ & \_ & = & \mathit{False} \\
\mathit{equal}\ \langle \mathrm{Prod}\ \alpha\ \beta \rangle & (x_1 \times x_2) & (y_1 \times y_2) & = & \quad \mathit{equal}\ \langle \alpha \rangle\ x_1\ y_1 \\
 & & & & \wedge\ \mathit{equal}\ \langle \beta \rangle\ x_2\ y_2\ .
\end{array}
$$

The type of the function is:

$$\mathit{equal}\ \langle a \rangle :: \quad a \rightarrow a \rightarrow \mathrm{Bool}$$

The function *equal* depends on itself, which is reflected in its type signature.

Generic equality in Dependency-style

Generic functions are defined by "pattern matching" on a type argument:

$$
\begin{array}{llll}
\mathit{equal}\ \langle \mathrm{Int} \rangle & & & = (==) \\
\mathit{equal}\ \langle \mathrm{Unit} \rangle & \mathit{Unit} & \mathit{Unit} & = \mathit{True} \\
\mathit{equal}\ \langle \mathrm{Sum}\ \alpha\ \beta \rangle & (\mathit{Inl}\ x) & (\mathit{Inl}\ y) & = \mathit{equal}\ \langle \alpha \rangle\ x\ y \\
\mathit{equal}\ \langle \mathrm{Sum}\ \alpha\ \beta \rangle & (\mathit{Inr}\ x) & (\mathit{Inr}\ y) & = \mathit{equal}\ \langle \beta \rangle\ x\ y \\
\mathit{equal}\ \langle \mathrm{Sum}\ \alpha\ \beta \rangle & \_ & \_ & = \mathit{False} \\
\mathit{equal}\ \langle \mathrm{Prod}\ \alpha\ \beta \rangle & (x_1 \times x_2) & (y_1 \times y_2) & = \quad \mathit{equal}\ \langle \alpha \rangle\ x_1\ y_1 \\
 & & & \quad\ \wedge \mathit{equal}\ \langle \beta \rangle\ x_2\ y_2 \,.
\end{array}
$$

The type of the function is:

$$\mathit{equal}\ \langle a \rangle :: (\mathit{equal}\ \langle a \rangle) \Rightarrow a \rightarrow a \rightarrow \mathrm{Bool}$$

The function *equal* depends on itself, which is reflected in its type signature.

Dependencies correspond to dictionary arguments

The case for Unit,

$$\textit{equal}\ \langle \text{Unit} \rangle \quad \textit{Unit} \quad \textit{Unit} \quad = \textit{True}\ ,$$

is translated to

$$\text{cp}(\textit{equal}, \text{Unit}) \quad \textit{Unit} \quad \textit{Unit} \quad = \textit{True}\ .$$

Dependencies correspond to dictionary arguments

The case for Prod,

$$\begin{aligned} \mathit{equal}\ \langle \mathrm{Prod}\ \alpha\ \beta \rangle\ (x_1 \times x_2)\ (y_1 \times y_2) = &\quad \mathit{equal}\ \langle \alpha \rangle\ x_1\ y_1 \\ &\wedge \mathit{equal}\ \langle \beta \rangle\ x_2\ y_2 \end{aligned}$$

is translated to

$$\begin{aligned} \mathsf{cp}(\mathit{equal}, \mathrm{Prod})\ \mathsf{cp}(\mathit{equal}, \alpha)\ \mathsf{cp}(\mathit{equal}, \beta) & \\ (x_1 \times x_2)\ (y_1 \times y_2) = &\quad \mathsf{cp}(\mathit{equal}, \alpha)\ x_1\ y_1 \\ &\wedge \mathsf{cp}(\mathit{equal}, \beta)\ x_2\ y_2 \end{aligned}$$

Dependencies correspond to dictionary arguments

The case for Prod,

$$\begin{aligned} \mathit{equal}\ \langle \mathrm{Prod}\ \alpha\ \beta \rangle\ (x_1 \times x_2)\ (y_1 \times y_2) = &\quad \mathit{equal}\ \langle \alpha \rangle\ x_1\ y_1 \\ &\wedge \mathit{equal}\ \langle \beta \rangle\ x_2\ y_2 \end{aligned}$$

is translated to

$$\begin{aligned} \mathsf{cp}(\mathit{equal}, \mathrm{Prod})\ \mathsf{cp}(\mathit{equal}, \alpha)\ \mathsf{cp}(\mathit{equal}, \beta) & \\ (x_1 \times x_2)\ (y_1 \times y_2) = &\quad \mathsf{cp}(\mathit{equal}, \alpha)\ x_1\ y_1 \\ &\wedge \mathsf{cp}(\mathit{equal}, \beta)\ x_2\ y_2 \end{aligned}$$

For each variable in the type pattern, there is a parameter explaining how to compute the dependency *equal*.

Dependencies in the types

Dependency constraints in the types make sure that no unresolved dependencies occur in a program.
For instance, the expression

$$\lambda(x_1 \times x_2)\ (y_1 \times y_2) \to \mathit{equal}\ \langle\alpha\rangle\ x_1\ x_2 \wedge \mathit{equal}\ \langle\beta\rangle\ y_1\ y_2$$

(the definition of *equal* $\langle \text{Prod} \rangle$) has type

$$\forall a\ b.(\mathit{equal}\ \langle\alpha\rangle :: a \to a \to \text{Bool}, \mathit{equal}\ \langle\beta\rangle :: b \to b \to \text{Bool}) \\ \quad \Rightarrow \text{Prod}\ a\ b \to \text{Prod}\ a\ b \to \text{Bool}$$

Dependency type signatures

$$equal\ \langle a \rangle :: (equal\ \langle a \rangle) \Rightarrow a \rightarrow a \rightarrow \mathrm{Bool}$$

There is an algorithm that computes from the type signature of a generic function the types that specific instances (may) have:

$$\begin{aligned}
&equal\ \langle \mathrm{Int} \rangle && :: \mathrm{Int} \rightarrow \mathrm{Int} \rightarrow \mathrm{Bool} \\
&equal\ \langle [\alpha] \rangle && :: \forall a.(equal\ \langle a \rangle :: a \rightarrow a \rightarrow \mathrm{Bool}) \\
& && \quad \Rightarrow [a] \rightarrow [a] \rightarrow \mathrm{Bool} \\
&equal\ \langle \mathrm{Sum}\ \alpha\ \beta \rangle && :: \forall a\ b.(equal\ \langle \alpha \rangle :: a \rightarrow a \rightarrow \mathrm{Bool}, \\
& && \qquad\quad equal\ \langle \beta \rangle :: b \rightarrow b \rightarrow \mathrm{Bool}) \\
& && \quad \Rightarrow \mathrm{Sum}\ a\ b \rightarrow \mathrm{Sum}\ a\ b \rightarrow \mathrm{Bool} \\
&equal\ \langle \mathrm{Fix}\ \varphi \rangle && :: \forall f.(equal\ \langle \varphi \rangle :: \forall a.\ (equal\ \langle \alpha \rangle :: a \rightarrow a \rightarrow \mathrm{Bool}) \\
& && \qquad\qquad\qquad\qquad\quad \Rightarrow f\ a \rightarrow f\ a \rightarrow \mathrm{Bool}) \\
& && \quad \Rightarrow \mathrm{Fix}\ f \rightarrow \mathrm{Fix}\ f \rightarrow \mathrm{Bool}
\end{aligned}$$

Dependencies and translation

Dependencies dictate how calls to generic functions are translated:

> $\mathit{equal}\ \langle \text{Sum Int Char} \rangle$
> $\quad \leadsto \mathsf{cp}(\mathit{equal}, \text{Sum})\ \mathsf{cp}(\mathit{equal}, \text{Int})\ \mathsf{cp}(\mathit{equal}, \text{Char})$

- Applications of generic functions can always be simplified to applications of components.
- Components are always parametrized over a function and a single type (constructor).
- Specialization of generic functions is compositional.

Dependencies and translation – contd.

The types of the components are ordered, flattened versions of the dependency types:

$$
\begin{aligned}
\mathit{equal}\ \langle \mathrm{Sum}\ \alpha\ \beta \rangle :: \forall a\ b.(&\mathit{equal}\ \langle \alpha \rangle :: a \to a \to \mathrm{Bool}, \\
&\mathit{equal}\ \langle \beta \rangle :: b \to b \to \mathrm{Bool}) \\
&\Rightarrow \mathrm{Sum}\ a\ b \to \mathrm{Sum}\ a\ b \to \mathrm{Bool}
\end{aligned}
$$

Dependencies and translation – contd.

The types of the components are ordered, flattened versions of the dependency types:

$$\begin{aligned}
\mathit{equal}\ \langle \mathrm{Sum}\ \alpha\ \beta \rangle &:: \forall a\ b.(\mathit{equal}\ \langle \alpha \rangle :: a \rightarrow a \rightarrow \mathrm{Bool}, \\
&\qquad\qquad \mathit{equal}\ \langle \beta \rangle :: b \rightarrow b \rightarrow \mathrm{Bool}) \\
&\qquad \Rightarrow \mathrm{Sum}\ a\ b \rightarrow \mathrm{Sum}\ a\ b \rightarrow \mathrm{Bool} \\
\mathsf{cp}(\mathit{equal}, \mathrm{Sum}) &:: \forall a\ b.(a \rightarrow a \rightarrow \mathrm{Bool}) \rightarrow (b \rightarrow b \rightarrow \mathrm{Bool}) \\
&\qquad \rightarrow \mathrm{Sum}\ a\ b \rightarrow \mathrm{Sum}\ a\ b \rightarrow \mathrm{Bool}
\end{aligned}$$

Dependencies and translation – contd.

The types of the components are ordered, flattened versions of the dependency types:

$$
\begin{aligned}
&\mathit{equal}\ \langle \mathrm{Sum}\ \alpha\ \beta \rangle &&:: \forall a\ b.(\mathit{equal}\ \langle \alpha \rangle :: a \to a \to \mathrm{Bool}, \\
& && \qquad\quad \mathit{equal}\ \langle \beta \rangle :: b \to b \to \mathrm{Bool}) \\
& && \quad \Rightarrow \mathrm{Sum}\ a\ b \to \mathrm{Sum}\ a\ b \to \mathrm{Bool} \\
&\mathsf{cp}(\mathit{equal}, \mathrm{Sum}) &&:: \forall a\ b.(a \to a \to \mathrm{Bool}) \to (b \to b \to \mathrm{Bool}) \\
& && \quad \to \mathrm{Sum}\ a\ b \to \mathrm{Sum}\ a\ b \to \mathrm{Bool} \\
&\mathit{equal}\ \langle \mathrm{Fix}\ \varphi \rangle &&:: \forall f.(\mathit{equal}\ \langle \varphi \rangle :: \forall a.\ (\mathit{equal}\ \langle \alpha \rangle :: a \to a \to \mathrm{Bool}) \\
& && \qquad\qquad\qquad\qquad \Rightarrow f\ a \to f\ a \to \mathrm{Bool}) \\
& && \quad \Rightarrow \mathrm{Fix}\ f \to \mathrm{Fix}\ f \to \mathrm{Bool}
\end{aligned}
$$

Dependencies and translation – contd.

The types of the components are ordered, flattened versions of the dependency types:

$$
\begin{aligned}
&\mathit{equal}\ \langle \mathrm{Sum}\ \alpha\ \beta \rangle :: \forall a\ b.(\mathit{equal}\ \langle \alpha \rangle :: a \to a \to \mathrm{Bool},\\
&\qquad\qquad\qquad\qquad \mathit{equal}\ \langle \beta \rangle :: b \to b \to \mathrm{Bool})\\
&\qquad\qquad\qquad \Rightarrow \mathrm{Sum}\ a\ b \to \mathrm{Sum}\ a\ b \to \mathrm{Bool}\\
&\mathsf{cp}(\mathit{equal}, \mathrm{Sum}) \ :: \forall a\ b.(a \to a \to \mathrm{Bool}) \to (b \to b \to \mathrm{Bool})\\
&\qquad\qquad\qquad \to \mathrm{Sum}\ a\ b \to \mathrm{Sum}\ a\ b \to \mathrm{Bool}\\
&\mathit{equal}\ \langle \mathrm{Fix}\ \varphi \rangle \ :: \forall f.(\mathit{equal}\ \langle \varphi \rangle :: \forall a.\ (\mathit{equal}\ \langle \alpha \rangle :: a \to a \to \mathrm{Bool})\\
&\qquad\qquad\qquad\qquad\qquad\qquad \Rightarrow f\ a \to f\ a \to \mathrm{Bool})\\
&\qquad\qquad\qquad \Rightarrow \mathrm{Fix}\ f \to \mathrm{Fix}\ f \to \mathrm{Bool}\\
&\mathsf{cp}(\mathit{equal}, \mathrm{Fix}) \ :: \forall f.(\forall a.(a \to a \to \mathrm{Bool}) \to f\ a \to f\ a \to \mathrm{Bool})\\
&\qquad\qquad\qquad \to \mathrm{Fix}\ f \to \mathrm{Fix}\ f \to \mathrm{Bool}
\end{aligned}
$$

Multiple dependencies

Of course, a function can not only depend on itself.

$$\begin{aligned} equal\ \langle \alpha \to \beta \rangle\ fx\ fy = equal\ \langle [\beta] \rangle\ & (map\ fx\ (enum\ \langle \alpha \rangle)) \\ & (map\ fy\ (enum\ \langle \alpha \rangle)) \end{aligned}$$

Multiple dependencies

Of course, a function can not only depend on itself.

$$\begin{aligned} \mathit{equal}\ \langle \alpha \to \beta \rangle\ \mathit{fx}\ \mathit{fy} = \mathit{equal}\ \langle [\beta] \rangle\ & (\mathit{map}\ \mathit{fx}\ (\mathit{enum}\ \langle \alpha \rangle)) \\ & (\mathit{map}\ \mathit{fy}\ (\mathit{enum}\ \langle \alpha \rangle)) \end{aligned}$$

For above case to be valid, the type of the generic function has to be adapted:

$$\mathit{equal}\ \langle a \rangle :: (\mathit{equal}\ \langle a \rangle, \mathit{enum}\ \langle a \rangle) \Rightarrow a \to a \to \mathrm{Bool}\ .$$

Dependencies scope over functions

- A dependency may be needed by only a single case, but it affects the type of the entire function.
- In the example of *equal* the function case dictates that *enum* must be a dependency of *equal*.
- However, we can write down the type of an application of a generic function using only the (single) type signature as information.

Dependencies affect translation

With dependencies on both *equal* and *enum*, the case

$$\begin{aligned} &\mathit{equal}\ \langle\alpha \to \beta\rangle\ \ \mathit{fx}\ \mathit{fy} = \mathit{equal}\ \langle[\beta]\rangle\ (\mathit{map}\ \mathit{fx}\ (\mathit{enum}\ \langle\alpha\rangle))\\ &\qquad\qquad\qquad\qquad\qquad\qquad\quad (\mathit{map}\ \mathit{fy}\ (\mathit{enum}\ \langle\alpha\rangle))\end{aligned}$$

is translated to

$$\begin{aligned} &\mathsf{cp}(\mathit{equal}, \to)\ \mathsf{cp}(\mathit{equal}, \alpha)\ \mathsf{cp}(\mathit{enum}, \alpha)\\ &\qquad\qquad\quad \mathsf{cp}(\mathit{equal}, \beta)\ \mathsf{cp}(\mathit{enum}, \beta)\\ &\qquad\qquad\qquad \mathit{fx}\ \mathit{fy} = \mathsf{cp}(\mathit{equal}, [\,])\ \mathsf{cp}(\mathit{equal}, \beta)\ \mathsf{cp}(\mathit{enum}, \beta)\\ &\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{map}\ \mathit{fx}\ \mathsf{cp}(\mathit{enum}, \alpha))\\ &\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{map}\ \mathit{fy}\ \mathsf{cp}(\mathit{enum}, \alpha))\end{aligned}$$

Dependencies affect translation – contd.

With type

$$equal \langle a \rangle :: (equal \langle a \rangle \qquad\qquad) \Rightarrow a \rightarrow a \rightarrow \text{Bool} \; ,$$

the case

$$\begin{aligned} equal \langle \text{Prod}\ \alpha\ \beta \rangle\ \ (x_1 \times x_2)\ (y_1 \times y_2) = &\quad equal \langle \alpha \rangle\ x_1\ y_1 \\ &\wedge\ equal \langle \beta \rangle\ x_2\ y_2 \end{aligned}$$

is translated to

$$\begin{aligned} \mathsf{cp}(equal, \text{Prod})\ &\mathsf{cp}(equal, \alpha) \\ &\mathsf{cp}(equal, \beta) \\ &\ \ (x_1 \times x_2)\ (y_1 \times y_2) = \quad \mathsf{cp}(equal, \alpha)\ x_1\ y_1 \\ &\qquad\qquad\qquad\qquad\quad \wedge\ \mathsf{cp}(equal, \beta)\ x_2\ y_2 \end{aligned}$$

Dependencies affect translation – contd.

With type

$$equal\ \langle a \rangle :: (equal\ \langle a \rangle, enum\ \langle a \rangle) \Rightarrow a \rightarrow a \rightarrow \text{Bool}\ ,$$

the case

$$\begin{aligned} equal\ \langle \text{Prod}\ \alpha\ \beta \rangle\ \ (x_1 \times x_2)\ (y_1 \times y_2) = &\ \ equal\ \langle \alpha \rangle\ x_1\ y_1 \\ \wedge &\ equal\ \langle \beta \rangle\ x_2\ y_2 \end{aligned}$$

is translated to

$$\begin{aligned} \mathsf{cp}(equal, \text{Prod})\ \mathsf{cp}(equal, \alpha)\ \mathsf{cp}(enum, \alpha) & \\ \mathsf{cp}(equal, \beta)\ \mathsf{cp}(enum, \beta) & \\ (x_1 \times x_2)\ (y_1 \times y_2) = &\ \ \mathsf{cp}(equal, \alpha)\ x_1\ y_1 \\ \wedge &\ \mathsf{cp}(equal, \beta)\ x_2\ y_2 \end{aligned}$$

Dependencies affect translation – contd.

At the call sites:

$$\begin{aligned} &\mathit{equal}\ \langle \text{Sum Int Char} \rangle \\ &\quad \leadsto \mathsf{cp}(\mathit{equal}, \text{Sum})\ \mathsf{cp}(\mathit{equal}, \text{Int}) \\ &\qquad\qquad\qquad\qquad\quad \mathsf{cp}(\mathit{equal}, \text{Char}) \end{aligned}$$

Dependencies affect translation – contd.

At the call sites:

$$
\begin{aligned}
&\mathit{equal}\ \langle \mathrm{Sum\ Int\ Char} \rangle \\
&\quad \rightsquigarrow \mathsf{cp}(\mathit{equal}, \mathrm{Sum})\ \mathsf{cp}(\mathit{equal}, \mathrm{Int})\quad \mathsf{cp}(\mathit{enum}, \mathrm{Int}) \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathsf{cp}(\mathit{equal}, \mathrm{Char})\ \mathsf{cp}(\mathit{enum}, \mathrm{Char})
\end{aligned}
$$

Disadvantages of function-scope dependencies

Even though dependencies allow generic functions that interact, they have a number of limitations.

- Open generic functions are unsatisfactory. Generic functions can be open, but new cases cannot introduce new dependencies, because the translation of existing cases changes.

Disadvantages of function-scope dependencies

Even though dependencies allow generic functions that interact, they have a number of limitations.

- Open generic functions are unsatisfactory. Generic functions can be open, but new cases cannot introduce new dependencies, because the translation of existing cases changes.
- The dependency relation is very inaccurate. Often, unneeded dependencies are passed around.

Disadvantages of function-scope dependencies

Even though dependencies allow generic functions that interact, they have a number of limitations.

- Open generic functions are unsatisfactory. Generic functions can be open, but new cases cannot introduce new dependencies, because the translation of existing cases changes.
- The dependency relation is very inaccurate. Often, unneeded dependencies are passed around.
- With other desirable generalizations – such as multiple type arguments for generic functions – function-scope dependencies become infeasible.

Local redefinition

If we have a case-insensitive comparison of strings

$$caseInsensitive\ x\ y = toUpper\ x == toUpper\ y\ ,$$

the fragment

$$\textbf{let}\ equal\ \langle\alpha\rangle = caseInsensitive$$
$$\textbf{in}\ \ equal\ \langle[\alpha]\rangle\ \texttt{"Lambda"}\ \texttt{"LAMBDA"}$$

evaluates to *True*.

Local redefinition

If we have a case-insensitive comparison of strings

$$caseInsensitive\ x\ y = toUpper\ x == toUpper\ y\ ,$$

the fragment

$$\begin{array}{l} \textbf{let}\ equal\ \langle\alpha\rangle = caseInsensitive \\ \quad\ \ enum\ \langle\alpha\rangle = ? \\ \textbf{in}\ \ equal\ \langle[\alpha]\rangle\ \texttt{"Lambda"}\ \texttt{"LAMBDA"} \end{array}$$

evaluates to *True*.

Local redefinition

If we have a case-insensitive comparison of strings

$$caseInsensitive\ x\ y = toUpper\ x == toUpper\ y\ ,$$

the fragment

$$\begin{aligned}
&\textbf{let}\ equal\ \langle\alpha\rangle = caseInsensitive \\
&\phantom{\textbf{let}}\ enum\ \langle\alpha\rangle = equal\ \langle \mathrm{Char} \rangle \\
&\textbf{in}\ \ equal\ \langle[\alpha]\rangle\ \texttt{"Lambda"}\ \texttt{"LAMBDA"}
\end{aligned}$$

evaluates to *True*.

Would dependencies by case be a solution?

$$
\begin{aligned}
&\mathit{equal}\ \langle a \rangle && :: a \rightarrow a \rightarrow \text{Bool} \\
&\mathit{equal}\ \langle \text{Prod}\ \alpha\ \beta \rangle &&= \{\text{-as before -}\} \\
&\mathit{equal}\ \langle \alpha \rightarrow \beta \rangle &&= \{\text{-as before -}\}
\end{aligned}
$$

Would dependencies by case be a solution?

$$
\begin{aligned}
&\mathit{equal}\ \langle a\rangle \quad :: a \to a \to \mathrm{Bool}\\
&(\mathit{equal}\ \langle\alpha\rangle, \mathit{equal}\ \langle\beta\rangle) \Rightarrow \mathit{equal}\ \langle \mathrm{Prod}\ \alpha\ \beta\rangle\\
&\mathit{equal}\ \langle \mathrm{Prod}\ \alpha\ \beta\rangle = \text{\{-as before -\}}\\
&(\mathit{equal}\ \langle\alpha\rangle, \mathit{enum}\ \langle\alpha\rangle, \mathit{equal}\ \langle\beta\rangle) \Rightarrow \mathit{equal}\ \langle \alpha \to \beta\rangle\\
&\mathit{equal}\ \langle \alpha \to \beta\rangle = \text{\{-as before -\}}
\end{aligned}
$$

Would dependencies by case be a solution?

$$\begin{aligned}
&\mathit{equal}\ \langle a\rangle \quad :: a \to a \to \mathrm{Bool}\\
&(\mathit{equal}\ \langle\alpha\rangle, \mathit{equal}\ \langle\beta\rangle) \Rightarrow \mathit{equal}\ \langle \mathrm{Prod}\ \alpha\ \beta\rangle\\
&\mathit{equal}\ \langle \mathrm{Prod}\ \alpha\ \beta\rangle = \text{\{-as before -\}}\\
&(\mathit{equal}\ \langle\alpha\rangle, \mathit{enum}\ \langle\alpha\rangle, \mathit{equal}\ \langle\beta\rangle) \Rightarrow \mathit{equal}\ \langle \alpha \to \beta\rangle\\
&\mathit{equal}\ \langle \alpha \to \beta\rangle = \text{\{-as before -\}}
\end{aligned}$$

- More like type classes.

Would dependencies by case be a solution?

$$
\begin{aligned}
&equal\ \langle a\rangle \quad :: a \to a \to \text{Bool}\\
&(equal\ \langle\alpha\rangle, equal\ \langle\beta\rangle) \Rightarrow equal\ \langle \text{Prod}\ \alpha\ \beta\rangle\\
&equal\ \langle \text{Prod}\ \alpha\ \beta\rangle = \ \{\text{-as before -}\}\\
&(equal\ \langle\alpha\rangle, enum\ \langle\alpha\rangle, equal\ \langle\beta\rangle) \Rightarrow equal\ \langle\alpha \to \beta\rangle\\
&equal\ \langle\alpha \to \beta\rangle \quad = \ \{\text{-as before -}\}
\end{aligned}
$$

- More like type classes.
- Less complicated types: no *enum* dependency if not actually needed, no *enum* dependency for β in the function case.

Would dependencies by case be a solution?

$$
\begin{aligned}
&equal\ \langle a\rangle \quad :: a \to a \to \text{Bool}\\
&(equal\ \langle\alpha\rangle, equal\ \langle\beta\rangle) \Rightarrow equal\ \langle\text{Prod}\ \alpha\ \beta\rangle\\
&equal\ \langle\text{Prod}\ \alpha\ \beta\rangle = \text{\{-as before -\}}\\
&(equal\ \langle\alpha\rangle, enum\ \langle\alpha\rangle, equal\ \langle\beta\rangle) \Rightarrow equal\ \langle\alpha \to \beta\rangle\\
&equal\ \langle\alpha \to \beta\rangle = \text{\{-as before -\}}
\end{aligned}
$$

- More like type classes.
- Less complicated types: no *enum* dependency if not actually needed, no *enum* dependency for β in the function case.
- Easier to use with other extensions such as multiple type arguments.

However …

…we can no longer determine the type of an application of a generic function using only the type signature of the function.

However . . .

. . . we can no longer determine the type of an application of a generic function using only the type signature of the function.

$$\mathit{equal}\ \langle \mathrm{Fix}\ \varphi \rangle \qquad = \ \ldots \mathit{foo}\ \langle \varphi\ \mathrm{Int} \rangle \ldots$$

is translated to

$$\mathsf{cp}(\mathit{equal}, \mathrm{Fix})\ \mathsf{cp}(\mathit{foo}, \varphi) = \ \ldots \mathsf{cp}(\mathit{foo}, \varphi)\ \mathsf{cp}(???, \mathrm{Int}) \ldots \ldots$$

However . . .

. . . we can no longer determine the type of an application of a generic function using only the type signature of the function.

$$equal\ \langle \mathrm{Fix}\ \varphi \rangle \qquad = \ \ldots foo\ \langle \varphi\ \mathrm{Int} \rangle \ldots$$

is translated to

$$\mathsf{cp}(equal, \mathrm{Fix})\ \mathsf{cp}(foo, \varphi) = \ \ldots \mathsf{cp}(foo, \varphi)\ \mathsf{cp}(???, \mathrm{Int}) \ldots\ldots$$

Abstracting from $\mathsf{cp}(foo, \varphi\ \mathrm{Int})$ rather than $\mathsf{cp}(foo, \varphi)$ breaks the compositionality of generic function components.

Conclusions

- Dependencies allow to write generic functions that use other generic functions.
- Dependencies that scope over the whole function have a number of disadvantages.
- Dependencies that scope over only one case seem to be the desired solution for many of the problems, but they introduce new limitations.
- It is desirable to solve this problem, because more liberal dependencies also seem helpful or even necessary to add other features such as multiple type arguments or complex type patterns.