Dependency-style Generic Haskell

Andres Löh, Dave Clarke, Johan Jeuring
Universiteit Utrecht
andres@cs.uu.nl

26th August 2003

Overview

- Classic Generic Haskell
  - Generic programming in Classic Generic Haskell
  - The problem with Classic Generic Haskell
- Solution: Dependency-style Generic Haskell
- Typing Dependency-style generic functions
- Examples
  - Classic generic functions
  - Generic traversal
- Comparison with type classes
- Conclusions

Classic Generic Haskell

- Generic Haskell = Haskell + generic functions (+ generic datatypes)
- generic = indexed by a type argument
- a generic function usually is defined inductively over the structure of datatypes
- thus, generic functions work for all types in a generic way
- Generic Haskell is implemented as a preprocessor that translates generic functions into Haskell
- translation proceeds by specialisation
- the theory is based on Ralf Hinze's several papers about generic programming in Haskell
- typical generic functions are: mapping, ordering, (de)coding, (un)parsing, generic traversals, operations on type-indexed datatypes

Programming in Classic Generic Haskell

A generic comparison function looks as follows:

$$\begin{array}{l}
\mathbf{data}\ \mathit{Ordering} = \mathit{LT} \mid \mathit{EQ} \mid \mathit{GT} \\
\mathbf{type}\ \mathit{Comp}\langle\!\langle\star\rangle\!\rangle\ t = t \to t \to \mathit{Ordering} \\
\mathbf{type}\ \mathit{Comp}\langle\!\langle\kappa \to \kappa'\rangle\!\rangle\ t = \forall a.\mathit{Comp}\langle\!\langle\kappa\rangle\!\rangle\ a \to \mathit{Comp}\langle\!\langle\kappa'\rangle\!\rangle\ (t\ a) \\
\mathit{comp}\langle t :: \kappa\rangle :: \mathit{Comp}\langle\!\langle\kappa\rangle\!\rangle\ t \\
\mathit{comp}\langle \mathit{Unit}\rangle\ \mathit{Unit}\ \mathit{Unit} = \mathit{EQ} \\
\mathit{comp}\langle \mathit{Sum}\rangle\ \mathit{comp}_a\ \mathit{comp}_b\ (\mathit{Inl}\ a_1)\ (\mathit{Inl}\ a_2) = \mathit{comp}_a\ a_1\ a_2 \\
\mathit{comp}\langle \mathit{Sum}\rangle\ \mathit{comp}_a\ \mathit{comp}_b\ (\mathit{Inl}\ _)\ (\mathit{Inr}\ _) = \mathit{LT} \\
\mathit{comp}\langle \mathit{Sum}\rangle\ \mathit{comp}_a\ \mathit{comp}_b\ (\mathit{Inr}\ _)\ (\mathit{Inl}\ _) = \mathit{GT} \\
\mathit{comp}\langle \mathit{Sum}\rangle\ \mathit{comp}_a\ \mathit{comp}_b\ (\mathit{Inr}\ b_1)\ (\mathit{Inr}\ b_2) = \mathit{comp}_b\ b_1\ b_2 \\
\mathit{comp}\langle \mathit{Prod}\rangle\ \mathit{comp}_a\ \mathit{comp}_b\ (a_1, b_1)\ (a_2, b_2) = \\
\quad \mathbf{case}\ \mathit{comp}_a\ a_1\ a_2\ \mathbf{of} \\
\qquad \mathit{EQ} \to \mathit{comp}_b\ b_1\ b_2 \\
\qquad r \quad\ \to r \\
\mathit{comp}\langle \mathit{Int}\rangle\ i_1\ i_2 = \mathit{compare}\ i_1\ i_2
\end{array}$$

A closer look

A **kind-indexed type** (kind argument in $\langle\!\langle \cdot \rangle\!\rangle$):

$$\textbf{type}\ \mathit{Comp}\langle\!\langle \star \rangle\!\rangle\ t = t \to t \to \mathit{Ordering}$$

The type of the function on normal (i.e. kind $\star$) type arguments.

$$\textbf{type}\ \mathit{Comp}\langle\!\langle \kappa \to \kappa' \rangle\!\rangle\ t = \forall a. \mathit{Comp}\langle\!\langle \kappa \rangle\!\rangle\ a \to \mathit{Comp}\langle\!\langle \kappa' \rangle\!\rangle\ (t\ a)$$

The type of the function on type constructors.

A closer look

A **kind-indexed type** (kind argument in $\langle\!\langle \cdot \rangle\!\rangle$):

$$\textbf{type}\ Comp\langle\!\langle \star \rangle\!\rangle\ t = t \to t \to Ordering$$

The type of the function on normal (i.e. kind $\star$) type arguments.

$$\textbf{type}\ Comp\langle\!\langle \kappa \to \kappa' \rangle\!\rangle\ t = \forall a. Comp\langle\!\langle \kappa \rangle\!\rangle\ a \to Comp\langle\!\langle \kappa' \rangle\!\rangle\ (t\ a)$$

The type of the function on type constructors.

A **type signature**:

$$comp\langle t :: \kappa \rangle :: Comp\langle\!\langle \kappa \rangle\!\rangle\ t$$

The type is assigned to the function.

A closer look — contd.

Multiple **cases**, for different **type patterns** (in $\langle \cdot \rangle$):

A closer look — contd.

Multiple **cases**, for different **type patterns** (in $\langle\cdot\rangle$):

$$comp\langle Unit\rangle \quad Unit \quad Unit \quad = EQ$$

The one-element type *Unit* is defined as follows:

data *Unit* = *Unit*

A closer look — contd.

Multiple **cases**, for different **type patterns** (in $\langle\cdot\rangle$):

$$comp\langle Unit\rangle \quad Unit \quad Unit = EQ$$

The one-element type *Unit* is defined as follows:

data $Unit = Unit$

$$\begin{aligned}
&comp\langle Sum\rangle\ comp_a\ comp_b\ (Inl\ a_1)\ (Inl\ a_2) = comp_a\ a_1\ a_2\\
&comp\langle Sum\rangle\ comp_a\ comp_b\ (Inl\ _\ )\ (Inr\ _\ ) = LT\\
&comp\langle Sum\rangle\ comp_a\ comp_b\ (Inr\ _\ )\ (Inl\ _\ ) = GT\\
&comp\langle Sum\rangle\ comp_a\ comp_b\ (Inr\ b_1)\ (Inr\ b_2) = comp_b\ b_1\ b_2
\end{aligned}$$

The type constructor *Sum* represents choice:

data $Sum\ a\ b = Inl\ a \mid Inr\ b$

- *Sum* is a type constructor of kind $\star \to \star \to \star$
- the cases for *Sum* get two comparison functions as arguments
- the definition of *comp* is written as a **catamorphism**

A closer look — contd.

$$
\begin{aligned}
&\mathit{comp}\langle \mathit{Prod}\rangle\ \mathit{comp}_a\ \mathit{comp}_b\ (a_1, b_1)\ \ (a_2, b_2)\ =\\
&\quad \mathbf{case}\ \mathit{comp}_a\ a_1\ a_2\ \mathbf{of}\\
&\qquad \mathit{EQ} \rightarrow \mathit{comp}_b\ b_1\ b_2\\
&\qquad r \quad\ \ \rightarrow r
\end{aligned}
$$

The type *Prod a b* contains pairs of *a*'s and *b*'s:

$$\mathbf{data}\ \mathit{Prod}\ a\ b = (a, b)$$

A closer look — contd.

$$\begin{array}{l} comp\langle Prod\rangle\ comp_a\ comp_b\ (a_1, b_1)\ (a_2, b_2) = \\ \quad \textbf{case}\ comp_a\ a_1\ a_2\ \textbf{of} \\ \qquad EQ \rightarrow comp_b\ b_1\ b_2 \\ \qquad r \quad\ \rightarrow r \end{array}$$

The type *Prod a b* contains pairs of *a*'s and *b*'s:

$$\textbf{data}\ Prod\ a\ b = (a, b)$$

Haskell datatypes can be represented by isomorphic datatypes that are built from *Unit*, *Sum*, *Prod*, plus type application, abstraction and recursion, and a few primitive types, such as *Int*:

$$comp\langle Int\rangle \qquad i_1 \qquad i_2 \qquad = compare\ i_1\ i_2$$

Here, *compare* denotes the standard comparison function defined in the prelude.

A closer look — contd.

$$
\begin{array}{l}
comp\langle Prod\rangle\ comp_a\ comp_b\ (a_1, b_1)\ \ (a_2, b_2)\ =\\
\quad \mathbf{case}\ comp_a\ a_1\ a_2\ \mathbf{of}\\
\qquad EQ \rightarrow comp_b\ b_1\ b_2\\
\qquad r \quad\ \rightarrow r
\end{array}
$$

The type *Prod a b* contains pairs of *a*'s and *b*'s:

$$\mathbf{data}\ Prod\ a\ b = (a, b)$$

Haskell datatypes can be represented by isomorphic datatypes that are built from *Unit*, *Sum*, *Prod*, plus type application, abstraction and recursion, and a few primitive types, such as *Int*:

$$comp\langle Int\rangle \qquad i_1 \qquad i_2 \qquad = compare\ i_1\ i_2$$

Here, *compare* denotes the standard comparison function defined in the prelude.

In this style of generic definition, the type patterns are always simple types or type constructors.

The virtue of having kind-indexed types

The generic function can be used on types of different kinds:

$$\textbf{data}\ \mathit{Tree}\ a = \mathit{Node}\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a) \mid \mathit{Leaf}\ a$$

$$t_1 = \mathit{Node}\ (\mathit{Leaf}\ 3)\ (\mathit{Leaf}\ 7)$$
$$t_2 = \mathit{Node}\ (\mathit{Leaf}\ 3)\ (\mathit{Leaf}\ 5)$$

$$\mathit{comp}\langle \mathit{Tree}\ \mathit{Int}\rangle :: \mathit{Tree}\ \mathit{Int} \to \mathit{Tree}\ \mathit{Int} \to \mathit{Ordering}$$
$$\mathit{comp}\langle \mathit{Tree}\rangle :: \forall a.(a \to a \to \mathit{Ordering}) \to (\mathit{Tree}\ a \to \mathit{Tree}\ a \to \mathit{Ordering})$$

$$\mathit{comp}\langle \mathit{Tree}\ \mathit{Int}\rangle\ t_1\ t_2 \rightsquigarrow \mathit{GT}$$
$$\mathit{comp}\langle \mathit{Tree}\rangle\ (\lambda x\ y \to \mathit{EQ})\ t_1\ t_2 \rightsquigarrow \mathit{EQ}$$

The virtue of having kind-indexed types

The generic function can be used on types of different kinds:

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

$$t_1 = \mathit{Node}\ (\mathit{Leaf}\ 3)\ (\mathit{Leaf}\ 7)$$
$$t_2 = \mathit{Node}\ (\mathit{Leaf}\ 3)\ (\mathit{Leaf}\ 5)$$

$$\mathit{comp}\langle \mathit{Tree\ Int}\rangle :: \mathit{Tree\ Int} \to \mathit{Tree\ Int} \to \mathit{Ordering}$$
$$\mathit{comp}\langle \mathit{Tree}\rangle :: \forall a.(a \to a \to \mathit{Ordering}) \to (\mathit{Tree}\ a \to \mathit{Tree}\ a \to \mathit{Ordering})$$

$$\mathit{comp}\langle \mathit{Tree\ Int}\rangle\ t_1\ t_2 \rightsquigarrow \mathit{GT}$$
$$\mathit{comp}\langle \mathit{Tree}\rangle\ (\lambda x\ y \to \mathit{EQ})\ t_1\ t_2 \rightsquigarrow \mathit{EQ}$$

Type application, abstraction, and recursion are interpreted as application, abstraction and recursion on the value level. For instance:

$$\mathit{comp}\langle \mathit{Tree\ Int}\rangle \equiv \mathit{comp}\langle \mathit{Tree}\rangle\ (\mathit{comp}\langle \mathit{Int}\rangle)$$

The virtue of having kind-indexed types

The generic function can be used on types of different kinds:

$\textbf{data}\ Tree\ a = Node\ (Tree\ a)\ (Tree\ a) \mid Leaf\ a$

$t_1 = Node\ (Leaf\ 3)\ (Leaf\ 7)$
$t_2 = Node\ (Leaf\ 3)\ (Leaf\ 5)$

$comp\langle Tree\ Int\rangle :: Tree\ Int \to Tree\ Int \to Ordering$
$comp\langle Tree\rangle :: \forall a.(a \to a \to Ordering) \to (Tree\ a \to Tree\ a \to Ordering)$

$comp\langle Tree\ Int\rangle\ t_1\ t_2 \rightsquigarrow GT$
$comp\langle Tree\rangle\ (\lambda x\ y \to EQ)\ t_1\ t_2 \rightsquigarrow EQ$

Type application, abstraction, and recursion are interpreted as application, abstraction and recursion on the value level. For instance:

$comp\langle Tree\ Int\rangle \equiv comp\langle Tree\rangle\ (comp\langle Int\rangle)$

Generic functions defined in this setting can be applied to type constructors of all kinds, to mutually recursive and nested datatypes!

A modified comparison function

Suppose we want to define a modified comparison function *lcomp* that implements efficient comparison of lists:

- first compare the lengths of the lists; only if they are equal, continue normally.

(→)

A modified comparison function

Suppose we want to define a modified comparison function *lcomp* that implements efficient comparison of lists:

- first compare the lengths of the lists; only if they are equal, continue normally.

All cases would be as for *comp*. In addition, there is one special case for the list type constructor [ ]:

$$
\begin{aligned}
&\mathit{lcomp}\langle[\,]\rangle\ \mathit{lcomp}_a\ \mathit{as}_1\ \mathit{as}_2 = \\
&\quad \textbf{case}\ \mathit{compare}\ (\mathit{length}\ \mathit{as}_1)\ (\mathit{length}\ \mathit{as}_2)\ \textbf{of} \\
&\qquad \mathit{EQ} \rightarrow \mathit{comp}_a\ \mathit{as}_1\ \mathit{as}_2 \\
&\qquad r \quad\ \rightarrow r
\end{aligned}
$$

We need to refer to comp_a but the catamorphic structure of the function definitions only gives us access to lcomp_a!

A modified comparison function

Suppose we want to define a modified comparison function *lcomp* that implements efficient comparison of lists:

- first compare the lengths of the lists; only if they are equal, continue normally.

All cases would be as for *comp*. In addition, there is one special case for the list type constructor $[\,]$:

$$
\begin{aligned}
&lcomp\langle[\,]\rangle\; lcomp_a\; as_1\; as_2 = \\
&\quad \textbf{case}\; compare\; (length\; as_1)\; (length\; as_2)\; \textbf{of} \\
&\qquad EQ \rightarrow comp_a\; as_1\; as_2 \\
&\qquad r \quad\; \rightarrow r
\end{aligned}
$$

We need to refer to $comp_a$ but the catamorphic structure of the function definitions only gives us access to $lcomp_a$!

If a generic function depends on other generic functions except itself, then it is difficult to express that in Classic Generic Haskell.

(→)

A workaround

One can tuple the function *lcomp* with *comp*:

$$\textbf{type}\ TComp\langle\langle\star\rangle\rangle\quad t = (t \to t \to Ordering, t \to t \to Ordering)$$
$$\textbf{type}\ TComp\langle\langle\kappa \to \kappa'\rangle\rangle\ t = \forall a. TComp\langle\langle\kappa\rangle\rangle\ a \to TComp\langle\langle\kappa'\rangle\rangle\ (t\ a)$$

$$tcomp\langle t :: \kappa\rangle :: TComp\langle\langle\kappa\rangle\rangle\ t$$

$\ldots$

$$\begin{aligned}
&tcomp\langle[\,]\rangle\ (lcomp_a, comp_a) = \\
&\quad (\lambda as_1\ as_2 \to \textbf{case}\ compare\ (length\ as_1)\ (length\ as_2)\ \textbf{of} \\
&\qquad\qquad\qquad EQ \to comp_a\ as_1\ as_2 \\
&\qquad\qquad\qquad r \quad\ \to r \\
&\quad , comp\langle List\rangle \\
&\quad)
\end{aligned}$$

Disadvantages of this approach:

- different aspects (different functions) become intertwined
- the definition is hard to read and complicated
- it does not scale well if more than two functions or mutually recursive functions are involved

Goal of Dependency-style Generic Haskell

We would like to write *lcomp* like this:

```
lcomp⟨Unit⟩        Unit      Unit      = EQ
lcomp⟨Sum δa δb⟩  (Inl a1)  (Inl a2)  = lcomp⟨δa⟩ a1 a2
lcomp⟨Sum δa δb⟩  (Inl _ )  (Inr _ )  = LT
lcomp⟨Sum δa δb⟩  (Inr _ )  (Inl _ )  = GT
lcomp⟨Sum δa δb⟩  (Inr b1)  (Inr b2)  = lcomp⟨δb⟩ b1 b2
lcomp⟨Prod δa δb⟩ (a1, b1)  (a2, b2)  = case lcomp⟨δa⟩ a1 a2 of
                                           EQ → lcomp⟨δb⟩ b1 b2
                                           r  → r
lcomp⟨Int⟩         i1        i2        = compare i1 i2
lcomp⟨[δa]⟩        as1       as2       = case compare (length as1) (length as2) of
                                           EQ → comp⟨δa⟩ as1 as2
                                           r  → r
```

(Type variables with δ are **scoped** over one case of the generic definition – we call them **dependency variables**.)

Goal of Dependency-style Generic Haskell

We would like to write *lcomp* like this:

$$
\begin{array}{llll}
\multicolumn{4}{l}{lcomp\langle\delta a\rangle\ \textbf{extends}\ comp\langle\delta a\rangle} \\
lcomp\langle[\delta a]\rangle & as_1 & as_2 & = \textbf{case}\ compare\ (length\ as_1)\ (length\ as_2)\ \textbf{of} \\
 & & & \quad EQ \rightarrow comp\langle\delta a\rangle\ as_1\ as_2 \\
 & & & \quad r \quad\ \rightarrow r
\end{array}
$$

(Type variables with δ are **scoped** over one case of the generic definition – we call them **dependency variables**.)

Goal of Dependency-style Generic Haskell

We would like to write *lcomp* like this:

$$
\begin{array}{l}
lcomp\langle\delta a\rangle\ \textbf{extends}\ comp\langle\delta a\rangle \\
lcomp\langle[\delta a]\rangle\quad as_1\quad as_2\quad = \textbf{case}\ compare\ (length\ as_1)\ (length\ as_2)\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad EQ \rightarrow comp\langle\delta a\rangle\ as_1\ as_2 \\
\qquad\qquad\qquad\qquad\qquad r \quad\ \rightarrow r
\end{array}
$$

(Type variables with δ are **scoped** over one case of the generic definition – we call them **dependency variables**.)

Have the better syntax using recursion explicitly, but keep all advantages of Classic Generic Haskell.

Dependency-style Generic Haskell

- The type patterns in the cases are now type constructors applied to dependency variables ($Sum\ \delta a\ \delta b$ instead of Sum).
- Explicit dictionaries are replaced by implicit dictionaries.
- The implicit dictionaries can not only hold the function that is defined, but other functions.
- These dependencies of one generic function on other generic functions are recorded in the types.

Explicit recursion, implicit dictionaries

$$
\begin{array}{lllll}
comp\langle Unit\rangle & & Unit & Unit & = EQ \\
comp\langle Sum\rangle & comp_a\ comp_b & (Inl\ a_1) & (Inl\ a_2) & = comp_a\ a_1\ a_2 \\
comp\langle Sum\rangle & comp_a\ comp_b & (Inl\ \_\ ) & (Inr\ \_\ ) & = LT \\
comp\langle Sum\rangle & comp_a\ comp_b & (Inr\ \_\ ) & (Inl\ \_\ ) & = GT \\
comp\langle Sum\rangle & comp_a\ comp_b & (Inr\ b_1) & (Inr\ b_2) & = comp_b\ b_1\ b_2 \\
comp\langle Prod\rangle & comp_a\ comp_b & (a_1, b_1) & (a_2, b_2) & =
\end{array}
$$

$$
\begin{array}{l}
\quad \textbf{case}\ comp_a\ a_1\ a_2\ \textbf{of} \\
\qquad EQ \rightarrow comp_b\ b_1\ b_2 \\
\qquad r \quad\ \rightarrow r
\end{array}
$$

$$
\begin{array}{lllll}
comp\langle Int\rangle & & i_1 & i_2 & = compare\ i_1\ i_2
\end{array}
$$

Explicit recursion, implicit dictionaries

$$
\begin{array}{llllll}
comp\langle Unit\rangle & & Unit & Unit & = EQ \\
comp\langle Sum\rangle & comp\langle\delta a\rangle\ comp\langle\delta b\rangle & (Inl\ a_1) & (Inl\ a_2) & = comp\langle\delta a\rangle\ a_1\ a_2 \\
comp\langle Sum\rangle & comp\langle\delta a\rangle\ comp\langle\delta b\rangle & (Inl\ \_\ ) & (Inr\ \_\ ) & = LT \\
comp\langle Sum\rangle & comp\langle\delta a\rangle\ comp\langle\delta b\rangle & (Inr\ \_\ ) & (Inl\ \_\ ) & = GT \\
comp\langle Sum\rangle & comp\langle\delta a\rangle\ comp\langle\delta b\rangle & (Inr\ b_1) & (Inr\ b_2) & = comp\langle\delta b\rangle\ b_1\ b_2 \\
comp\langle Prod\rangle & comp\langle\delta a\rangle\ comp\langle\delta b\rangle & (a_1, b_1) & (a_2, b_2) & = \\
\quad \mathbf{case}\ comp\langle\delta a\rangle\ a_1\ a_2\ \mathbf{of} \\
\qquad EQ \rightarrow comp\langle\delta b\rangle\ b_1\ b_2 \\
\qquad r \quad\ \rightarrow r \\
comp\langle Int\rangle & & i_1 & i_2 & = compare\ i_1\ i_2
\end{array}
$$

We rename the dictionary arguments.

Explicit recursion, implicit dictionaries

$$
\begin{array}{llll}
\mathit{comp}\langle \mathit{Unit}\rangle & \mathit{Unit} & \mathit{Unit} & = \mathit{EQ} \\
\mathit{comp}\langle \mathit{Sum}\ \delta a\ \delta b\rangle\ \mathit{comp}\langle \delta a\rangle\ \mathit{comp}\langle \delta b\rangle & (\mathit{Inl}\ a_1) & (\mathit{Inl}\ a_2) & = \mathit{comp}\langle \delta a\rangle\ a_1\ a_2 \\
\mathit{comp}\langle \mathit{Sum}\ \delta a\ \delta b\rangle\ \mathit{comp}\langle \delta a\rangle\ \mathit{comp}\langle \delta b\rangle & (\mathit{Inl}\ \_) & (\mathit{Inr}\ \_) & = \mathit{LT} \\
\mathit{comp}\langle \mathit{Sum}\ \delta a\ \delta b\rangle\ \mathit{comp}\langle \delta a\rangle\ \mathit{comp}\langle \delta b\rangle & (\mathit{Inr}\ \_) & (\mathit{Inl}\ \_) & = \mathit{GT} \\
\mathit{comp}\langle \mathit{Sum}\ \delta a\ \delta b\rangle\ \mathit{comp}\langle \delta a\rangle\ \mathit{comp}\langle \delta b\rangle & (\mathit{Inr}\ b_1) & (\mathit{Inr}\ b_2) & = \mathit{comp}\langle \delta b\rangle\ b_1\ b_2 \\
\mathit{comp}\langle \mathit{Prod}\ \delta a\ \delta b\rangle\ \mathit{comp}\langle \delta a\rangle\ \mathit{comp}\langle \delta b\rangle & (a_1, b_1) & (a_2, b_2) & = \\
\quad \mathbf{case}\ \mathit{comp}\langle \delta a\rangle\ a_1\ a_2\ \mathbf{of} & & & \\
\qquad \mathit{EQ} \rightarrow \mathit{comp}\langle \delta b\rangle\ b_1\ b_2 & & & \\
\qquad r \quad\ \ \rightarrow r & & & \\
\mathit{comp}\langle \mathit{Int}\rangle & i_1 & i_2 & = \mathit{compare}\ i_1\ i_2
\end{array}
$$

We add variables to the type arguments.

Explicit recursion, implicit dictionaries

$$
\begin{array}{llllll}
comp\langle Unit\rangle & & Unit & Unit & = EQ \\
comp\langle Sum\ \delta a\ \delta b\rangle & \text{\{- (\ldots) implicit -\}} & (Inl\ a_1) & (Inl\ a_2) & = comp\langle \delta a\rangle\ a_1\ a_2 \\
comp\langle Sum\ \delta a\ \delta b\rangle & \text{\{- (\ldots) implicit -\}} & (Inl\ \_\ ) & (Inr\ \_\ ) & = LT \\
comp\langle Sum\ \delta a\ \delta b\rangle & \text{\{- (\ldots) implicit -\}} & (Inr\ \_\ ) & (Inl\ \_\ ) & = GT \\
comp\langle Sum\ \delta a\ \delta b\rangle & \text{\{- (\ldots) implicit -\}} & (Inr\ b_1) & (Inr\ b_2) & = comp\langle \delta b\rangle\ b_1\ b_2 \\
comp\langle Prod\ \delta a\ \delta b\rangle & \text{\{- (\ldots) implicit -\}} & (a_1, b_1) & (a_2, b_2) & = \\
\quad \mathbf{case}\ comp\langle \delta a\rangle\ a_1\ a_2\ \mathbf{of} & & & & \\
\qquad EQ \rightarrow comp\langle \delta b\rangle\ b_1\ b_2 & & & & \\
\qquad r \quad\ \rightarrow r & & & & \\
comp\langle Int\rangle & & i_1 & i_2 & = compare\ i_1\ i_2
\end{array}
$$

We forget the dictionary arguments.

- ➔ This definition is in the desired format, but can be interpreted in the same way as the Classic definition.
- ➔ Type arguments are type constructors, fully applied to dependency variables.

What about the types?

The dependencies are recorded in the types.

$$comp\langle Sum\ \delta a\ \delta b\rangle\ (Inl\ a_1)\ (Inl\ a_2) = comp\langle \delta a\rangle\ a_1\ a_2$$
$$comp\langle Sum\ \delta a\ \delta b\rangle\ (Inl\ \_\ )\ (Inr\ \_\ ) = LT$$
$$comp\langle Sum\ \delta a\ \delta b\rangle\ (Inr\ \_\ )\ (Inl\ \_\ ) = GT$$
$$comp\langle Sum\ \delta a\ \delta b\rangle\ (Inr\ b_1)\ (Inr\ b_2) = comp\langle \delta b\rangle\ b_1\ b_2$$

For instance, the right hand sides of the sum case have this type:

$$\forall a\ b.(comp\langle \delta a\rangle :: a \to a \to Ordering, comp\langle \delta b\rangle :: b \to b \to Ordering)$$
$$\Rightarrow Sum\ a\ b \to Sum\ a\ b \to Ordering$$

Actually, these four types are instances of the type given above:

$$\forall a\ b.(comp\langle \delta a\rangle :: a \to a \to Ordering) \Rightarrow Sum\ a\ b \to Sum\ a\ b \to Ordering$$
$$\forall a\ b.\qquad Sum\ a\ b \to Sum\ a\ b \to Ordering$$
$$\forall a\ b.\qquad Sum\ a\ b \to Sum\ a\ b \to Ordering$$
$$\forall a\ b.(comp\langle \delta b\rangle :: b \to b \to Ordering) \Rightarrow Sum\ a\ b \to Sum\ a\ b \to Ordering$$

What about the types? – contd.

Dependencies are introduced whenever a type argument with one or more dependency variables is used. For instance, $comp\langle \delta a \rangle$:

$$comp\langle \delta a \rangle :: \quad a \to a \to Ordering$$

What about the types? – contd.

Dependencies are introduced whenever a type argument with one or more dependency variables is used. For instance, $comp\langle \delta a \rangle$:

$$comp\langle \delta a \rangle :: (comp\langle \delta a \rangle :: a \to a \to Ordering) \Rightarrow a \to a \to Ordering$$

What about the types? – contd.

Dependencies are introduced whenever a type argument with one or more dependency variables is used. For instance, $comp\langle \delta a \rangle$:

$$comp\langle \delta a \rangle :: (comp\langle \delta a \rangle :: a \to a \to Ordering) \Rightarrow a \to a \to Ordering$$

It turns out that this type contains sufficient type information for the generic function:

$$(comp\langle \delta a \rangle :: a \to a \to Ordering) \Rightarrow a \to a \to Ordering$$

What about the types? – contd.

Dependencies are introduced whenever a type argument with one or more dependency variables is used. For instance, $comp\langle\delta a\rangle$:

$$comp\langle\delta a\rangle :: (comp\langle\delta a\rangle :: a \to a \to Ordering) \Rightarrow a \to a \to Ordering$$

It turns out that this type contains sufficient type information for the generic function:

$$\langle\delta a\rangle\ a \mapsto$$
$$(comp\langle\delta a\rangle :: a \to a \to Ordering) \Rightarrow a \to a \to Ordering$$

What about the types? – contd.

Dependencies are introduced whenever a type argument with one or more dependency variables is used. For instance, $comp\langle\delta a\rangle$:

$$comp\langle\delta a\rangle :: (comp\langle\delta a\rangle :: a \to a \to Ordering) \Rightarrow a \to a \to Ordering$$

It turns out that this type contains sufficient type information for the generic function:

$$comp\langle t\rangle :: (\mathbf{generalize}\ \langle\delta a\rangle\ a \mapsto (comp\langle\delta a\rangle :: a \to a \to Ordering) \Rightarrow a \to a \to Ordering)\ t$$

What about the types? – contd.

Dependencies are introduced whenever a type argument with one or more dependency variables is used. For instance, $comp\langle\delta a\rangle$:

$$comp\langle\delta a\rangle :: (comp\langle\delta a\rangle :: a \to a \to Ordering) \Rightarrow a \to a \to Ordering$$

It turns out that this type contains sufficient type information for the generic function:

$$\begin{array}{l} comp\langle t\rangle :: (\textbf{generalize}\ \langle\delta a\rangle\ a \mapsto \\ \quad (comp\langle\delta a\rangle :: a \to a \to Ordering) \Rightarrow a \to a \to Ordering)\ t \end{array}$$

From this type signature, the following types can be computed automatically:

$$\begin{array}{lll} comp\langle[Int]\rangle & :: & [Int] \to [Int] \to Ordering \\ comp\langle[\delta a]\rangle & :: & (comp\langle\delta a\rangle :: a \to a \to Ordering) \\ & & \Rightarrow [a] \to [a] \to Ordering \\ comp\langle Sum\ \delta a\ \delta b\rangle & :: & (comp\langle\delta a\rangle :: a \to a \to Ordering \\ & & ,comp\langle\delta b\rangle :: b \to b \to Ordering) \\ & & \Rightarrow Sum\ a\ b \to Sum\ a\ b \to Ordering \end{array}$$

Using dependency-style functions

- The call $comp\langle Int\rangle$ refers to the case for Int in the definition.
- In Classic Generic Haskell, $comp\langle Tree\rangle$ expects an extra argument. The call $comp\langle Tree\ Int\rangle$ is the same as $comp\langle Tree\rangle\ (comp\langle Int\rangle)$.

Using dependency-style functions

- The call $comp\langle Int\rangle$ refers to the case for Int in the definition.
- In Classic Generic Haskell, $comp\langle Tree\rangle$ expects an extra argument. The call $comp\langle Tree\ Int\rangle$ is the same as $comp\langle Tree\rangle\ (comp\langle Int\rangle)$.
- Now, $comp\langle Tree\ \delta a\rangle$ has a **dependency** on $comp\langle \delta a\rangle :: a \rightarrow a \rightarrow Ordering$. This dependency can be satisfied in a special let-binding:

$$comp\langle Tree\ \delta a\rangle\ (Node\ (Leaf\ 3)\ (Leaf\ 7))\ (Node\ (Leaf\ 3)\ (Leaf\ 5))$$

Using dependency-style functions

- The call $comp\langle Int\rangle$ refers to the case for Int in the definition.
- In Classic Generic Haskell, $comp\langle Tree\rangle$ expects an extra argument. The call $comp\langle Tree\ Int\rangle$ is the same as $comp\langle Tree\rangle\ (comp\langle Int\rangle)$.
- Now, $comp\langle Tree\ \delta a\rangle$ has a **dependency** on $comp\langle \delta a\rangle :: a \to a \to Ordering$. This dependency can be satisfied in a special let-binding:

$$comp\langle \delta a\rangle$$
$$comp\langle Tree\ \delta a\rangle\ (Node\ (Leaf\ 3)\ (Leaf\ 7))\ (Node\ (Leaf\ 3)\ (Leaf\ 5))$$

Using dependency-style functions

- The call $comp\langle Int\rangle$ refers to the case for Int in the definition.
- In Classic Generic Haskell, $comp\langle Tree\rangle$ expects an extra argument. The call $comp\langle Tree\ Int\rangle$ is the same as $comp\langle Tree\rangle\ (comp\langle Int\rangle)$.
- Now, $comp\langle Tree\ \delta a\rangle$ has a **dependency** on $comp\langle \delta a\rangle :: a \to a \to Ordering$. This dependency can be satisfied in a special let-binding:

```
deplet comp⟨δa⟩ = (λx y → EQ) in
  comp⟨Tree δa⟩ (Node (Leaf 3) (Leaf 7)) (Node (Leaf 3) (Leaf 5))
```

Using dependency-style functions

- The call $comp\langle Int\rangle$ refers to the case for Int in the definition.
- In Classic Generic Haskell, $comp\langle Tree\rangle$ expects an extra argument. The call $comp\langle Tree\ Int\rangle$ is the same as $comp\langle Tree\rangle\ (comp\langle Int\rangle)$.
- Now, $comp\langle Tree\ \delta a\rangle$ has a **dependency** on $comp\langle \delta a\rangle :: a \to a \to Ordering$. This dependency can be satisfied in a special let-binding:

$$\mathbf{deplet}\ comp\langle \delta a\rangle = (\lambda x\ y \to EQ)\ \mathbf{in}$$
$$comp\langle Tree\ \delta a\rangle\ (Node\ (Leaf\ 3)\ (Leaf\ 7))\ (Node\ (Leaf\ 3)\ (Leaf\ 5))$$
$$\leadsto EQ$$

Using dependency-style functions

- The call $comp\langle Int\rangle$ refers to the case for *Int* in the definition.
- In Classic Generic Haskell, $comp\langle Tree\rangle$ expects an extra argument. The call $comp\langle Tree\ Int\rangle$ is the same as $comp\langle Tree\rangle\ (comp\langle Int\rangle)$.
- Now, $comp\langle Tree\ \delta a\rangle$ has a **dependency** on $comp\langle \delta a\rangle :: a \to a \to Ordering$. This dependency can be satisfied in a special let-binding:

$$\textbf{deplet}\ comp\langle \delta a\rangle = (\lambda x\ y \to EQ)\ \textbf{in}$$
$$comp\langle Tree\ \delta a\rangle\ (Node\ (Leaf\ 3)\ (Leaf\ 7))\ (Node\ (Leaf\ 3)\ (Leaf\ 5))$$
$$\rightsquigarrow EQ$$

- The call $comp\langle Tree\ Int\rangle$ now is the same as

$$\textbf{deplet}\ comp\langle \delta a\rangle = comp\langle Int\rangle\ \textbf{in}\ comp\langle Tree\ \delta a\rangle$$

Multiple dependencies

The function *lcomp* (with the special case for lists) depends on both *lcomp* and *comp*:

$$
\begin{aligned}
&lcomp\langle Prod\ \delta a\ \delta b\rangle\ (a_1, b_1)\ (a_2, b_2) = \textbf{case}\ lcomp\langle \delta a\rangle\ a_1\ a_2\ \textbf{of} \\
&\qquad EQ \rightarrow lcomp\langle \delta b\rangle\ b_1\ b_2 \\
&\qquad r \quad \rightarrow r \\
&\ldots \\
&lcomp\langle [\delta a]\rangle\ as_1\ as_2 = \textbf{case}\ compare\ (length\ as_1)\ (length\ as_2)\ \textbf{of} \\
&\qquad EQ \rightarrow comp\langle \delta a\rangle\ as_1\ as_2 \\
&\qquad r \quad \rightarrow r
\end{aligned}
$$

Multiple dependencies

The function *lcomp* (with the special case for lists) depends on both *lcomp* and *comp*:

$$
\begin{aligned}
&lcomp\langle Prod\ \delta a\ \delta b\rangle\ (a_1, b_1)\ (a_2, b_2) = \mathbf{case}\ lcomp\langle \delta a\rangle\ a_1\ a_2\ \mathbf{of} \\
&\qquad EQ \to lcomp\langle \delta b\rangle\ b_1\ b_2 \\
&\qquad r \quad \to r \\
&\ldots \\
&lcomp\langle [\delta a]\rangle\ as_1\ as_2 = \mathbf{case}\ compare\ (length\ as_1)\ (length\ as_2)\ \mathbf{of} \\
&\qquad EQ \to comp\langle \delta a\rangle\ as_1\ as_2 \\
&\qquad r \quad \to r
\end{aligned}
$$

$$
\begin{aligned}
&lcomp\langle t\rangle :: (\mathbf{generalize}\ \langle \delta a\rangle\ a \mapsto \\
&\quad (comp\langle \delta a\rangle :: a \to a \to Ordering \\
&\quad , lcomp\langle \delta a\rangle :: a \to a \to Ordering) \Rightarrow a \to a \to Ordering)\ t
\end{aligned}
$$

Multiple dependencies

The function *lcomp* (with the special case for lists) depends on both *lcomp* and *comp*:

$$
\begin{array}{llll}
lcomp\langle Prod\ \delta a\ \delta b\rangle\ (a_1, b_1)\ (a_2, b_2) & = \mathbf{case}\ lcomp\langle \delta a\rangle\ a_1\ a_2\ \mathbf{of} \\
& \quad EQ \rightarrow lcomp\langle \delta b\rangle\ b_1\ b_2 \\
& \quad r \quad\ \rightarrow r \\
\ldots \\
lcomp\langle [\delta a]\rangle \quad as_1 \quad as_2 & = \mathbf{case}\ compare\ (length\ as_1)\ (length\ as_2)\ \mathbf{of} \\
& \quad EQ \rightarrow comp\langle \delta a\rangle\ as_1\ as_2 \\
& \quad r \quad\ \rightarrow r
\end{array}
$$

$$
\begin{array}{l}
lcomp\langle t\rangle :: (\mathbf{generalize}\ \langle \delta a\rangle\ a \mapsto \\
\quad (comp\langle \delta a\rangle\ :: a \rightarrow a \rightarrow Ordering \\
\quad , lcomp\langle \delta a\rangle :: a \rightarrow a \rightarrow Ordering) \Rightarrow a \rightarrow a \rightarrow Ordering)\ t
\end{array}
$$

$$
\begin{array}{l}
lcomp\langle Tree\ Int\rangle \equiv \\
\quad \mathbf{deplet}\ comp\langle \delta a\rangle\ = comp\langle Int\rangle \\
\qquad\qquad lcomp\langle \delta a\rangle = lcomp\langle Int\rangle \\
\quad \mathbf{in}\ lcomp\langle Tree\ \delta a\rangle
\end{array}
$$

Traversal example

```
data Compiler   = C Name [Package Maintainer]
data Package a  = P Name a [Feature] [Package a]
data Maintainer = M Name Affiliation
                | Unmaintained
data Feature    = F String
type Name       = String
type Affiliation = String
```

Possible tasks:

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

Check if something is maintained

```
data Compiler   = C Name [Package Maintainer]
data Package a  = P Name a [Feature] [Package a]
data Maintainer = M Name Affiliation
                | Unmaintained
data Feature    = F String
type Name       = String
type Affiliation = String
```

```
unmaintained⟨δa⟩ extends crush⟨δa⟩ False (∨)
unmaintained⟨Package δa⟩ (P _ a _ _) = unmaintained⟨δa⟩ a
unmaintained⟨Maintainer⟩ m           = case m of Unmaintained → True
                                                 _            → False
```

Check if something is maintained

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

$$
\begin{aligned}
&\mathit{unmaintained}\langle\delta a\rangle\ \textbf{extends}\ \mathit{crush}\langle\delta a\rangle\ \mathit{False}\ (\vee)\\
&\mathit{unmaintained}\langle\mathit{Package}\ \delta a\rangle\ \ (P\ \_\ a\ \_\ \_) = \mathit{unmaintained}\langle\delta a\rangle\ a\\
&\mathit{unmaintained}\langle\mathit{Maintainer}\rangle\ m \qquad\quad = \textbf{case}\ m\ \textbf{of}\ \mathit{Unmaintained} \rightarrow \mathit{True}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \_ \qquad\qquad\qquad \rightarrow \mathit{False}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{unmaintained}\langle t\rangle :: (\textbf{generalize}\ \langle\delta a\rangle\ a \mapsto\\
&\quad (\mathit{unmaintained}\langle\delta a\rangle :: a \rightarrow \mathit{Bool}) \Rightarrow a \rightarrow \mathit{Bool})\ t\\
&\mathit{crush}\langle t\rangle :: \forall b.(\textbf{generalize}\ \langle\delta a\rangle\ a \mapsto\\
&\quad (\mathit{crush}\langle\delta a\rangle :: b \rightarrow (b \rightarrow b \rightarrow b) \rightarrow a \rightarrow b) \Rightarrow b \rightarrow (b \rightarrow b \rightarrow b) \rightarrow a \rightarrow b)\ t
\end{aligned}
$$

Assign a new maintainer to a structure

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

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{assign}\langle \delta a\rangle\ m\ \textbf{extends}\ \mathit{id}\langle \delta a\rangle} \\
\mathit{assign}\langle \mathit{Package}\ \delta a\rangle\ (\mathit{P\ name\ a\ fts\ pkgs}) & = & \mathit{P\ name}\ (\mathit{assign}\langle \delta a\rangle\ a)\ \mathit{fts\ pkgs} \\
\mathit{assign}\langle \mathit{Maintainer}\rangle\ \_ & = & m
\end{array}
$$

Assign a new maintainer to a structure

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

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{assign}\langle\delta a\rangle\ m\ \textbf{extends}\ \mathit{id}\langle\delta a\rangle} \\
\mathit{assign}\langle\mathit{Package}\ \delta a\rangle\ (\mathit{P\ name\ a\ fts\ pkgs}) & = & \mathit{P\ name}\ (\mathit{assign}\langle\delta a\rangle\ a)\ \mathit{fts\ pkgs} \\
\mathit{assign}\langle\mathit{Maintainer}\rangle\ \_ & = & m
\end{array}
$$

$$
\begin{array}{ll}
\mathit{assign}\langle t\rangle & :: (\textbf{generalize}\ \langle\delta a\rangle\ a \mapsto (\mathit{assign}\langle\delta a\rangle :: a \to a) \Rightarrow a \to a)\ t \\
\mathit{id}\langle t\rangle & :: (\textbf{generalize}\ \langle\delta a\rangle\ a \mapsto (\mathit{id}\langle\delta a\rangle :: a \to a) \Rightarrow a \to a)\ t
\end{array}
$$

Reassign suitable packages to me

$$
\begin{aligned}
& \mathit{gpreassign}\langle \delta a\rangle\ \textbf{extends}\ \mathit{id}\langle \delta a\rangle \\
& \mathit{gpreassign}\langle \mathit{Package}\ \delta a\rangle\ p@(P\ \mathit{name}\ a\ \mathit{fts}\ \mathit{pkgs}) \\
& \quad \mid \texttt{"generic programming"} \in \mathit{fts} \wedge \mathit{unmaintained}\langle \mathit{Package}\ \delta a\rangle \\
& \qquad\qquad = \mathit{assign}\langle \mathit{Package}\ \delta a\rangle\ (M\ \texttt{"Andres"}\ \texttt{"UU"})\ p' \\
& \quad \mid \mathit{otherwise} = p' \\
& \quad \textbf{where}\ p' = P\ \mathit{name}\ a\ \mathit{fts}\ (\mathit{gpreassign}\langle [\mathit{Package}\ \delta a]\rangle\ \mathit{pkgs})
\end{aligned}
$$

Reassign suitable packages to me

$$
\begin{array}{l}
\mathit{gpreassign}\langle \delta a\rangle\ \textbf{extends}\ \mathit{id}\langle \delta a\rangle \\
\mathit{gpreassign}\langle \mathit{Package}\ \delta a\rangle\ p@(P\ \mathit{name}\ a\ \mathit{fts}\ \mathit{pkgs}) \\
\quad \mid \texttt{"generic programming"} \in \mathit{fts} \wedge \mathit{unmaintained}\langle \mathit{Package}\ \delta a\rangle \\
\qquad\qquad\qquad = \mathit{assign}\langle \mathit{Package}\ \delta a\rangle\ (M\ \texttt{"Andres"}\ \texttt{"UU"})\ p' \\
\quad \mid \mathit{otherwise} = p' \\
\quad \textbf{where}\ p' = P\ \mathit{name}\ a\ \mathit{fts}\ (\mathit{gpreassign}\langle [\mathit{Package}\ \delta a]\rangle\ \mathit{pkgs})
\end{array}
$$

This time, there are three dependencies:

$$
\begin{array}{l}
\mathit{gpreassign}\langle t\rangle :: (\textbf{generalize}\ \langle \delta a\rangle\ a \mapsto \\
\quad (\mathit{unmaintained}\langle \delta a\rangle :: a \to \mathit{Bool} \\
\quad ,\mathit{assign}\langle \delta a\rangle \qquad\quad :: a \to a \\
\quad ,\mathit{gpreassign}\langle \delta a\rangle \quad\ :: a \to a) \Rightarrow a \to a)\ t
\end{array}
$$

Summary of Dependency style

- In the definitions of generic functions, the type patterns now are type construnctors applied to dependency variables.
- Calls to generic functions with type arguments containing dependency variables now give rise to dependency constraints.
- Dependency constraints can be satisfied by means of a **deplet** construct.

Comparison with type classes

- One function per class.
- Based on dependency variables.
- Dependency constraints can be locally instantiated.
- Type of the contraint varies with the kind of the variable; constraints can be nested:

$$\textbf{data}\ \mathit{Fix}\ f = \mathit{In}\ f\ (\mathit{Fix}\ f)$$

$$\mathit{comp}\langle \mathit{Fix}\ \delta f\rangle ::$$

$$\mathit{Fix}\ f \to \mathit{Fix}\ f \to \mathit{Ordering}$$

Comparison with type classes

- One function per class.
- Based on dependency variables.
- Dependency constraints can be locally instantiated.
- Type of the contraint varies with the kind of the variable; constraints can be nested:

$$
\begin{aligned}
&\textbf{data}\ \mathit{Fix}\ f = \mathit{In}\ f\ (\mathit{Fix}\ f)\\
&\mathit{comp}\langle \mathit{Fix}\ \delta f\rangle ::\\
&\quad (\mathit{comp}\langle \delta f \quad \rangle ::\\
&\qquad\quad f\ a \to f\ a \to \mathit{Ordering})\\
&\quad \Rightarrow \mathit{Fix}\ f \to \mathit{Fix}\ f \to \mathit{Ordering}
\end{aligned}
$$

Comparison with type classes

- One function per class.
- Based on dependency variables.
- Dependency constraints can be locally instantiated.
- Type of the contraint varies with the kind of the variable; constraints can be nested:

$$\textbf{data}\ \mathit{Fix}\ f = \mathit{In}\ f\ (\mathit{Fix}\ f)$$

$$\begin{aligned}
&\mathit{comp}\langle \mathit{Fix}\ \delta f\rangle :: \\
&\quad (\mathit{comp}\langle \delta f \quad \rangle :: \\
&\qquad (\mathit{comp}\langle \quad \rangle :: a \to a \to \mathit{Ordering}) \\
&\qquad \Rightarrow f\ a \to f\ a \to \mathit{Ordering}) \\
&\quad \Rightarrow \mathit{Fix}\ f \to \mathit{Fix}\ f \to \mathit{Ordering}
\end{aligned}$$

Comparison with type classes

- One function per class.
- Based on dependency variables.
- Dependency constraints can be locally instantiated.
- Type of the contraint varies with the kind of the variable; constraints can be nested:

$$\mathbf{data}\ \mathit{Fix}\ f = \mathit{In}\ f\ (\mathit{Fix}\ f)$$

$$\begin{aligned}
&\mathit{comp}\langle \mathit{Fix}\ \delta f\rangle :: \\
&\quad (\mathit{comp}\langle \delta f\ \delta a\rangle :: \\
&\qquad (\mathit{comp}\langle \delta a\rangle :: a \to a \to \mathit{Ordering}) \\
&\qquad \Rightarrow f\ a \to f\ a \to \mathit{Ordering}) \\
&\quad \Rightarrow \mathit{Fix}\ f \to \mathit{Fix}\ f \to \mathit{Ordering}
\end{aligned}$$

Conclusions

- Using Dependency-style Generic Haskell shifts programming complexity from the programmer to the compiler; the programmer can write functions in the more natural, "explicit" style, with named type arguments.
- Using multiple, possibly mutually recursive generic functions becomes possible.
- Nothing of the power of Classic Generic Haskell is lost.
- With Dependency-style syntax, it is easier to support even more classes of generic functions:
  - functions with higher base kind or nested type patterns

    $$\mathit{poly}\langle \Lambda \delta a.\delta a \rangle = \ldots$$
    $$\mathit{generic}\langle [[\delta a]] \rangle = \ldots$$

  - functions that involve type-indexed datatypes
- More future work: inferring the dependency constraints in the declaration of a generic function automatically.