The Right *Kind* of Generic Programming

José Pedro Magalhães

University of Oxford

September 9, 2012

Kinds!

GHC 7.6 is out! This means we now have:

Kinds!

GHC 7.6 is out! This means we now have:

- User-definable kinds through datatype promotion

Kinds!

GHC 7.6 is out! This means we now have:

- User-definable kinds through datatype promotion
- Kind polymorphism

Kinds!

GHC 7.6 is out! This means we now have:

- User-definable kinds through datatype promotion
- Kind polymorphism
- Type-level literals (more or less)

Kinds!

GHC 7.6 is out! This means we now have:

- User-definable kinds through datatype promotion
- Kind polymorphism
- Type-level literals (more or less)
- Lots of feature requests for 7.8!

Kinds!

GHC 7.6 is out! This means we now have:

- User-definable kinds through datatype promotion
- Kind polymorphism
- Type-level literals (more or less)
- Lots of feature requests for 7.8!

In this talk I'll show how we can use the new features to improve generic programming in GHC...

Kinds!

GHC 7.6 is out! This means we now have:

- User-definable kinds through datatype promotion
- Kind polymorphism
- Type-level literals (more or less)
- Lots of feature requests for 7.8!

In this talk I'll show how we can use the new features to improve generic programming in GHC... and also how we can *fail* to improve generic programming with them.

Promotion

For the standard Peano naturals and length-indexed vectors example, please see the paper. Here we shall delve right into universes!

```
data Universe star = U
                   | K star
                   | I
                   | C MetaCon (Universe star)
                   | (Universe star) :+: (Universe star)
                   | (Universe star) :×: (Universe star)
```

Promotion

For the standard Peano naturals and length-indexed vectors example, please see the paper. Here we shall delve right into universes!

$$
\begin{aligned}
\textbf{data}\ \mathit{Universe\ star} = {} & \mathit{U} \\
\mid {} & \mathit{K}\ \mathit{star} \\
\mid {} & \mathit{I} \\
\mid {} & \mathit{C}\ \mathit{MetaCon}\ (\mathit{Universe\ star}) \\
\mid {} & (\mathit{Universe\ star})\ :+:\ (\mathit{Universe\ star}) \\
\mid {} & (\mathit{Universe\ star})\ :\times:\ (\mathit{Universe\ star})
\end{aligned}
$$

Conceptual example:

$$\mathit{ListRep} = (\mathit{U}\ :+:\ ((\mathit{K}\ \mathit{Int})\ :\times:\ \mathit{I})\ ::\ \mathit{Universe\ Int})$$

Promotion

For the standard Peano naturals and length-indexed vectors example, please see the paper. Here we shall delve right into universes!

```
data Universe star = U
                   | K star
                   | I
                   | C MetaCon (Universe star)
                   | (Universe star) :+: (Universe star)
                   | (Universe star) :×: (Universe star)
```

Conceptual example:

```
ListRep = (U :+: ((K Int) :×: I) :: Universe Int)   -- nonsense
```

Promotion

For the standard Peano naturals and length-indexed vectors example, please see the paper. Here we shall delve right into universes!

$$
\begin{aligned}
\textbf{data}\ \mathit{Universe\ star} = {} & \mathit{U} \\
\mid {} & \mathit{K}\ \mathit{star} \\
\mid {} & \mathit{I} \\
\mid {} & \mathit{C}\ \mathit{MetaCon}\ (\mathit{Universe\ star}) \\
\mid {} & (\mathit{Universe\ star})\ :+:\ (\mathit{Universe\ star}) \\
\mid {} & (\mathit{Universe\ star})\ :\times:\ (\mathit{Universe\ star})
\end{aligned}
$$

Conceptual example:

$$\textbf{type}\ \mathit{ListRep} = (\mathit{U}\ :+:\ ((\mathit{K}\ \mathit{Int})\ :\times:\ \mathit{I}) :: \mathit{Universe}\ \star)$$

Interpretation

The interpretation defines the inhabitants of the universe:

$$
\begin{aligned}
&\textbf{data } [\![\, \upsilon :: \mathit{Universe} \star \,]\!]\ (\tau :: \star) \textbf{ where} \\
&\quad U' :: && [\![\, U \,]\!]\ \tau \\
&\quad K' :: \alpha && \to [\![\, (K\ \alpha) \,]\!]\ \tau \\
&\quad I' :: \tau && \to [\![\, I \,]\!]\ \tau \\
&\quad C' :: (\mathit{Constructor}\ \nu) \Rightarrow [\![\, \alpha \,]\!]\ \tau && \to [\![\, (C\ (MC\ \nu\ \phi\ \rho)\ \alpha) \,]\!]\ \tau \\
&\quad L' :: [\![\, \alpha \,]\!]\ \tau && \to [\![\, \alpha :+: \beta \,]\!]\ \tau \\
&\quad R' :: [\![\, \beta \,]\!]\ \tau && \to [\![\, \alpha :+: \beta \,]\!]\ \tau \\
&\quad \times' :: [\![\, \alpha \,]\!]\ \tau && \to [\![\, \beta \,]\!]\ \tau \to [\![\, \alpha :\times: \beta \,]\!]\ \tau
\end{aligned}
$$

Interpretation

The interpretation defines the inhabitants of the universe:

$$
\begin{array}{l}
\textbf{data}\ [\![\, \upsilon :: \mathit{Universe}\ \star \,]\!]\ (\tau :: \star)\ \textbf{where} \\
\quad U' :: \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad [\![\, U \,]\!]\ \tau \\
\quad K' :: \alpha \qquad\qquad\qquad\qquad\qquad\qquad \to [\![\, (K\ \alpha) \,]\!]\ \tau \\
\quad I' \ :: \tau \qquad\qquad\qquad\qquad\qquad\qquad \to [\![\, I \,]\!]\ \tau \\
\quad C' :: (\mathit{Constructor}\ \nu) \Rightarrow [\![\, \alpha \,]\!]\ \tau \to [\![\, (C\ (MC\ \nu\ \phi\ \rho)\ \alpha) \,]\!]\ \tau \\
\quad L' \ :: [\![\, \alpha \,]\!]\ \tau \qquad\qquad\qquad\qquad \to [\![\, \alpha :+: \beta \,]\!]\ \tau \\
\quad R' :: [\![\, \beta \,]\!]\ \tau \qquad\qquad\qquad\qquad \to [\![\, \alpha :+: \beta \,]\!]\ \tau \\
\quad \times' :: [\![\, \alpha \,]\!]\ \tau \qquad\qquad\qquad\qquad \to [\![\, \beta \,]\!]\ \tau \to [\![\, \alpha :\times: \beta \,]\!]\ \tau
\end{array}
$$

Conceptual example:

$$
\begin{array}{l}
\mathit{list}_1, \mathit{list}_2 :: [\![\, \mathit{ListRep} \,]\!]\ [\mathit{Int}] \\
\mathit{list}_1 = L'\ U' \\
\mathit{list}_2 = R'\ ((K'\ 0) \times' (I'\ []))
\end{array}
$$

Meta-information

Recall the type of C':

$$C' :: (Constructor\ v) \Rightarrow [\![\, \alpha \,]\!]\ \tau \rightarrow [\![\, (C\ (MC\ v\ \phi\ \rho)\ \alpha) \,]\!]\ \tau$$

We encode meta-information at the type level:

```
data MetaCon = MC Symbol Fixity Bool
data Fixity  = Prefix | Infix Assoc Nat
data Assoc   = LeftAssoc | RightAssoc | NotAssoc

data Proxy σ = Proxy
class Constructor (v :: Symbol) where
  conName :: Proxy v → String
  conFixity :: Proxy v → Fixity
  conFixity = const Prefix
  conIsRecord :: Proxy v → Bool
  conIsRecord = const False
```

Conversions to/from the generic representation

We use a type class to define the representable types:

$$
\begin{aligned}
&\textbf{class } \mathit{Regular}\ (\alpha :: \star)\ \textbf{where} \\
&\quad \textbf{type } \mathit{PF}\ \alpha :: \mathit{Universe} \star \\
&\quad \mathit{from} :: \alpha \to [\![\mathit{PF}\ \alpha]\!]\ \alpha \\
&\quad \mathit{to} \quad :: [\![\mathit{PF}\ \alpha]\!]\ \alpha \to \alpha
\end{aligned}
$$

The kind of the PF type family now clearly establishes that only representation types make up a valid representation.

Example encoding

This is how we can encode lists:

```
instance Regular [α] where
  type PF [α] =  (C (MC "[]" (Infix RightAssoc 5) False) U)
             :+: (C (MC ":" Prefix False) ((K α) :×: I))
  from []      = L' (C' U')
  from (h : t) = R' (C' (×' (K' h) (I' t)))
  to (L' (C' U'))                    = []
  to (R' (C' (×' (K' h) (I' t))))    = h : t

instance Constructor "[]" where conName _ = "[]"
instance Constructor ":"  where conName _ = ":"
```

Note the type-level symbols and naturals!

Improving Typeable

Let's now look at how to improve the *Typeable* class. The current situation:

$$\textbf{data}\ \mathit{TypeRep}$$

$$\textbf{class}\ \mathit{Typeable}\ (\alpha :: \star)\ \textbf{where}$$
$$\quad \mathit{typeOf} :: \alpha \to \mathit{TypeRep}$$

$$\textbf{class}\ \mathit{Typeable}_1\ (\phi :: \star \to \star)\ \textbf{where}$$
$$\quad \mathit{typeOf}_1 :: \phi\ \alpha \to \mathit{TypeRep}$$

For *Maybe*, for example, we need two instances:

$$\textbf{instance}\ \mathit{Typeable}_1\ \mathit{Maybe} \ldots$$

$$\textbf{instance}\ (\mathit{Typeable}\ \alpha) \Rightarrow \mathit{Typeable}\ (\mathit{Maybe}\ \alpha) \ldots$$

Kind-polymorphic Typeable

A single, kind-polymorphic *Typeable* class:

$$\textbf{class}\ \mathit{Typeable}\ (\phi :: \kappa)\ \textbf{where}$$
$$\quad \mathit{typeRep} :: \mathit{Proxy}\ \phi \to \mathit{TypeRep}$$

Kind-polymorphic Typeable

A single, kind-polymorphic *Typeable* class:

$$\textbf{class}\ \mathit{Typeable}\ (\phi :: \kappa)\ \textbf{where}$$
$$\quad \mathit{typeRep} :: \mathit{Proxy}\ \phi \to \mathit{TypeRep}$$

A single, kind-polymorphic instance for applied types:

$$\textbf{instance}\ (\mathit{Typeable}\ (\phi :: \kappa_1 \to \kappa_2), \mathit{Typeable}\ (\alpha :: \kappa_1)) \Rightarrow \mathit{Typeable}\ (\phi\ \alpha) \ldots$$

Kind-polymorphic Typeable

A single, kind-polymorphic *Typeable* class:

$$\textbf{class}\ \mathit{Typeable}\ (\phi :: \kappa)\ \textbf{where}$$
$$\quad \mathit{typeRep} :: \mathit{Proxy}\ \phi \to \mathit{TypeRep}$$

A single, kind-polymorphic instance for applied types:

$$\textbf{instance}\ (\mathit{Typeable}\ (\phi :: \kappa_1 \to \kappa_2), \mathit{Typeable}\ (\alpha :: \kappa_1)) \Rightarrow \mathit{Typeable}\ (\phi\ \alpha) \ldots$$

Functions for backwards compatibility:

$$\mathit{typeOf} :: \forall (\alpha :: \star).\ \mathit{Typeable}\ \alpha \Rightarrow \alpha \to \mathit{TypeRep}$$
$$\mathit{typeOf}\ \_ = \mathit{typeRep}\ (\mathit{Proxy} :: \mathit{Proxy}\ \alpha)$$
$$\mathit{typeOf}_1 :: \forall (\phi :: \star \to \star)(\alpha :: \star).\ \mathit{Typeable}\ \phi \Rightarrow \phi\ \alpha \to \mathit{TypeRep}$$
$$\mathit{typeOf}_1\ \_ = \mathit{typeRep}\ (\mathit{Proxy} :: \mathit{Proxy}\ \phi)$$

Kind-polymorphic SYB? I

The current situation:

$$
\begin{aligned}
&\textbf{newtype}\ \mathit{Result}\ (\alpha :: \star) = \mathit{Result}\ \mathit{Int}\\
&\mathit{example} :: \forall \alpha.\ (\mathit{Data}\ \alpha, \mathit{Typeable}\ \alpha) \Rightarrow \mathit{Result}\ \alpha\\
&\mathit{example} = \mathit{general}\ `\mathit{ext}_1`\ \mathit{maybe}_1\ `\mathit{ext}_0`\ \mathit{maybe}_0\\
&\quad \textbf{where}\ \mathit{general} :: \mathit{Result}\ \alpha\\
&\qquad\qquad \mathit{general} = \mathit{Result}\ 0\\
&\qquad\qquad \mathit{maybe}_0 :: \mathit{Result}\ (\mathit{Maybe}\ \mathit{Int})\\
&\qquad\qquad \mathit{maybe}_0 = \mathit{Result}\ 1\\
&\qquad\qquad \mathit{maybe}_1 :: \forall \chi.\ \mathit{Result}\ (\mathit{Maybe}\ \chi)\\
&\qquad\qquad \mathit{maybe}_1 = \mathit{Result}\ 2
\end{aligned}
$$

The *example* function returns:

- $\mathit{Result}\ 0$ when its type is instantiated to $\mathit{Result}\ \mathit{Int}$;
- $\mathit{Result}\ 1$ when its type is instantiated to $\mathit{Result}\ (\mathit{Maybe}\ \mathit{Int})$;
- $\mathit{Result}\ 2$ when its type is instantiated to $\mathit{Result}\ (\mathit{Maybe}\ \mathit{Char})$.

Kind-polymorphic SYB? II

The adhoc extension functions rely on a number of primitives, one per kind:

$$ext_0 :: (Typeable\ (\alpha :: \star), Typeable\ (\beta :: \star)) \Rightarrow \phi\ \alpha \to \phi\ \beta \to \phi\ \alpha$$

$$ext_0\ def\ ext = maybe\ def\ id\ (gcast\ ext)$$

$$ext_1 :: (Data\ \alpha, Typeable_1\ \psi) \Rightarrow \phi\ \alpha \to (\forall \beta.\ Data\ \beta \Rightarrow \phi\ (\psi\ \beta)) \to \phi\ \alpha$$

$$ext_1\ def\ ext = maybe\ def\ id\ (dataCast_1\ ext)$$

Kind-polymorphic SYB? III

gcast uses *typeOf*:

```
gcast :: (Typeable (α :: ⋆), Typeable β) ⇒ φ α → Maybe (φ β)
gcast x = r where
  r = if typeOf (getArg x) ≡ typeOf (getArg (fromJust r))
    then Just $ unsafeCoerce x
    else Nothing
  getArg :: φ α → α
  getArg = ⊥
```

$dataCast_1$ comes from the *Data* class:

```
class Data α where
  dataCast₁ :: Typeable₁ ψ
            ⇒ (∀β. Data β ⇒ φ (ψ β)) → Maybe (φ α)
  dataCast₁ _ = Nothing
```

Kind-polymorphic SYB? IV

Types of kind $\star \to \star$ define $dataCast_1$ to be $gcast_1$, which uses $typeOf_1$:

$$
\begin{aligned}
&gcast_1 \;::\; (Typeable_1\ (\psi :: \star \to \star), Typeable_1\ \psi') \\
&\qquad \Rightarrow \phi\ (\psi\ \alpha) \to Maybe\ (\phi\ (\psi'\ \alpha)) \\
&gcast_1\ x = r\ \textbf{where} \\
&\quad r = \textbf{if}\ typeOf_1\ (getArg\ x) \equiv typeOf_1\ (getArg\ (fromJust\ r)) \\
&\qquad \textbf{then}\ Just\ \$\ unsafeCoerce\ x \\
&\qquad \textbf{else}\ Nothing \\
&\quad getArg :: \phi\ \alpha \to \alpha \\
&\quad getArg = \bot
\end{aligned}
$$

Can we collapse $gcast$, $gcast_1$, etc. into a single definition?

Kind-polymorphic SYB? V

Using kind polymorphism:

$$
\begin{aligned}
&\mathit{gcast} :: \forall(\alpha :: \kappa_1)(\beta :: \kappa_2)(\phi :: \kappa_1 \to \star)(\psi :: \kappa_2 \to \star).\\
&\qquad (\mathit{Typeable}\ \alpha, \mathit{Typeable}\ \beta) \Rightarrow \phi\ \alpha \to \mathit{Maybe}\ (\psi\ \beta)\\
&\mathit{gcast}\ x = r\ \textbf{where}\\
&\quad r = \textbf{if}\ \mathit{typeRep}\ (\mathit{getArg}\ x) \equiv \mathit{typeRep}\ (\mathit{getArg}\ (\mathit{fromJust}\ r))\\
&\qquad \textbf{then}\ \mathit{Just}\ \$\ \mathit{unsafeCoerce}\ x\\
&\qquad \textbf{else}\ \mathit{Nothing}\\
&\quad \mathit{getArg} :: \forall \phi\, \alpha.\ \phi\ \alpha \to \mathit{Proxy}\ \alpha\\
&\quad \mathit{getArg} = \bot
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{ext} :: \forall(\alpha :: \kappa_1)(\beta :: \kappa_2)(\phi :: \kappa_1 \to \star)(\psi :: \kappa_2 \to \star).\\
&\qquad (\mathit{Typeable}\ \alpha, \mathit{Typeable}\ \beta) \Rightarrow \phi\ \alpha \to \psi\ \beta \to \phi\ \alpha\\
&\mathit{ext}\ \mathit{def} = \mathit{maybe}\ \mathit{def}\ \mathit{id} \circ \mathit{gcast}
\end{aligned}
$$

Kind-polymorphic SYB? VI

We can redefine *example* using the new *ext*:

$$
\begin{aligned}
&\textbf{newtype}\ \mathit{Result}\ (\alpha :: \kappa) = \mathit{Result}\ \mathit{Int}\\
&\mathit{example} :: \forall \alpha.\ (\mathit{Typeable}\ \alpha) \Rightarrow \mathit{Result}\ \alpha\\
&\mathit{example} = \mathit{general}\ \text{`}\mathit{ext}\text{`}\ \mathit{maybe}_0\ \text{`}\mathit{ext}\text{`}\ \mathit{maybe}_1\\
&\quad \textbf{where}\ \mathit{general} :: \mathit{Result}\ \alpha\\
&\qquad\qquad \mathit{general} = \mathit{Result}\ 0\\
&\qquad\qquad \mathit{maybe}_0 :: \mathit{Result}\ (\mathit{Maybe}\ \mathit{Int})\\
&\qquad\qquad \mathit{maybe}_0 = \mathit{Result}\ 1\\
&\qquad\qquad \mathit{maybe}_1 :: \mathit{Result}\ \mathit{Maybe}\\
&\qquad\qquad \mathit{maybe}_1 = \mathit{Result}\ 2
\end{aligned}
$$

Kind-polymorphic SYB? VI

We can redefine *example* using the new *ext*:

$$
\begin{aligned}
&\textbf{newtype}\ \mathit{Result}\ (\alpha :: \kappa) = \mathit{Result}\ \mathit{Int}\\
&\mathit{example} :: \forall \alpha.\ (\mathit{Typeable}\ \alpha) \Rightarrow \mathit{Result}\ \alpha\\
&\mathit{example} = \mathit{general}\ \text{`}\mathit{ext}\text{`}\ \mathit{maybe}_0\ \text{`}\mathit{ext}\text{`}\ \mathit{maybe}_1\\
&\quad \textbf{where}\ \mathit{general} :: \mathit{Result}\ \alpha\\
&\qquad\qquad \mathit{general} = \mathit{Result}\ 0\\
&\qquad\qquad \mathit{maybe}_0 :: \mathit{Result}\ (\mathit{Maybe}\ \mathit{Int})\\
&\qquad\qquad \mathit{maybe}_0 = \mathit{Result}\ 1\\
&\qquad\qquad \mathit{maybe}_1 :: \mathit{Result}\ \mathit{Maybe}\\
&\qquad\qquad \mathit{maybe}_1 = \mathit{Result}\ 2
\end{aligned}
$$

. . . but unfortunately this example is not general enough. How can we inspect the input values?

Generic programming in GHC

The current approach:

$$
\begin{aligned}
&\textbf{class } \mathit{Generic}\ \alpha \textbf{ where} \\
&\quad \textbf{type } \mathit{Rep}\ \alpha :: \star \to \star \\
&\quad \mathit{from} \ :: \alpha \to (\mathit{Rep}\ \alpha)\ \chi \\
&\quad \mathit{to} \ :: (\mathit{Rep}\ \alpha)\ \chi \to \alpha
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{class } \mathit{Generic}_1\ \phi \textbf{ where} \\
&\quad \textbf{type } \mathit{Rep}_1\ \phi :: \star \to \star \\
&\quad \mathit{from}_1 :: \phi\ \alpha \to \mathit{Rep}_1\ \phi\ \alpha \\
&\quad \mathit{to}_1 \ :: \mathit{Rep}_1\ \phi\ \alpha \to \phi\ \alpha
\end{aligned}
$$

Two classes, supporting only up to one parameter. Kind polymorphism to the rescue?

First attempt

Encode also the list of parameters associated with a type:

$$
\begin{aligned}
&\textbf{class } \mathit{Generic}\ (\alpha :: \star)\ \textbf{where} \\
&\quad \textbf{type } \mathit{Rep}\ \alpha :: \mathit{Universe}\ \star \\
&\quad \textbf{type } \mathit{Es}\quad \alpha :: [\star] \\
&\quad \mathit{from} :: \alpha \to [\![\, \mathit{Rep}\ \alpha \,]\!]\ (\mathit{Es}\ \alpha) \\
&\quad \mathit{to}\quad :: [\![\, \mathit{Rep}\ \alpha \,]\!]\ (\mathit{Es}\ \alpha) \to \alpha
\end{aligned}
$$

Possible universe and interpretation

$$\begin{aligned}\textbf{data}\ \mathit{Universe\ star} = {} & U \mid K\ \mathit{star} \mid P\ \mathit{Nat} \\ & \mid (\mathit{Universe\ star}) :+: (\mathit{Universe\ star}) \\ & \mid (\mathit{Universe\ star}) :\times: (\mathit{Universe\ star})\end{aligned}$$

$$\begin{aligned}\textbf{data}\ [\![\, \upsilon :: \mathit{Universe} \star \,]\!]\ (\tau :: [\star]) :: \star\ \textbf{where} \\ U' &:: [\![\, U \,]\!]\ \tau \\ K' &:: \alpha \to [\![\, (K\ \alpha) \,]\!]\ \tau \\ P' &:: N_{th}\ \tau\ \nu \to [\![\, (P\ \nu) \,]\!]\ \tau \\ L' &:: [\![\, \alpha \,]\!]\ \tau \to [\![\, \alpha :+: \beta \,]\!]\ \tau \\ R' &:: [\![\, \beta \,]\!]\ \tau \to [\![\, \alpha :+: \beta \,]\!]\ \tau \\ \times' &:: [\![\, \alpha \,]\!]\ \tau \to [\![\, \beta \,]\!]\ \tau \to [\![\, \alpha :\times: \beta \,]\!]\ \tau\end{aligned}$$

Possible universe and interpretation

$$\begin{aligned}
&\textbf{data}\ \mathit{Universe\ star} = \mathit{U} \mid \mathit{K\ star} \mid \mathit{P\ Nat} \\
&\qquad\qquad\qquad\qquad \mid (\mathit{Universe\ star}) :+: (\mathit{Universe\ star}) \\
&\qquad\qquad\qquad\qquad \mid (\mathit{Universe\ star}) :\times: (\mathit{Universe\ star})
\end{aligned}$$

$$\begin{aligned}
&\textbf{data}\ [\![\, \upsilon :: \mathit{Universe} \star \,]\!]\ (\tau :: [\star]) :: \star\ \textbf{where} \\
&\quad \mathit{U}' :: \qquad\qquad\qquad\qquad\quad [\![\, \mathit{U} \,]\!]\ \tau \\
&\quad \mathit{K}' :: \alpha \qquad\qquad\qquad\quad \to [\![\, (\mathit{K}\ \alpha) \,]\!]\ \tau \\
&\quad \mathit{P}' :: \mathit{N_{th}}\ \tau\ \nu \qquad\quad \to [\![\, (\mathit{P}\ \nu) \,]\!]\ \tau \\
&\quad \mathit{L}' :: [\![\, \alpha \,]\!]\ \tau \qquad\quad \to [\![\, \alpha :+: \beta \,]\!]\ \tau \\
&\quad \mathit{R}' :: [\![\, \beta \,]\!]\ \tau \qquad\quad \to [\![\, \alpha :+: \beta \,]\!]\ \tau \\
&\quad \times' :: [\![\, \alpha \,]\!]\ \tau \to [\![\, \beta \,]\!]\ \tau \to [\![\, \alpha :\times: \beta \,]\!]\ \tau
\end{aligned}$$

We need a type-level lookup function on lists:

$$\begin{aligned}
&\textbf{type family}\ \mathit{N_{th}}\ (\tau :: [\star])\ (\nu :: \mathit{Nat}) :: \star \\
&\textbf{type instance}\ \mathit{N_{th}}\ (\alpha : \beta)\ \mathit{Ze} \quad = \alpha \\
&\textbf{type instance}\ \mathit{N_{th}}\ (\alpha : \beta)\ (\mathit{Su}\ \nu) = \mathit{N_{th}}\ \beta\ \nu
\end{aligned}$$

Problems

The universe shown will unfortunately not work:

Problems

The universe shown will unfortunately not work:

- Safety of N_{th}: cannot use promoted vectors

Problems

The universe shown will unfortunately not work:

- Safety of N_{th}: cannot use promoted vectors
- Composition?...
  - We want to be able to *fmap* over parameters
  - It's easy when we only have one parameter
  - But what if we have two or three? E.g. composing [] with *Either*
  - We need more explicit control over how to "connect" inputs and outputs

Sledgehammer solution: indexed functors

Universe codes indexed by their input and output:

$$
\begin{aligned}
\textbf{data}\ \mathit{Code}\ \iota\ o = {} & \mathit{Z} \\
\mid {} & \mathit{U} \\
\mid {} & \mathit{I}\ \iota \\
\mid {} & !\ (\mathit{Code}\ \iota\ o)\ o \\
\mid {} & (\mathit{Code}\ \iota\ o) :+: (\mathit{Code}\ \iota\ o) \\
\mid {} & (\mathit{Code}\ \iota\ o) :\times: (\mathit{Code}\ \iota\ o) \\
\mid {} & \forall \nu.\ (\mathit{Code}\ \nu\ o) :\cdot: (\mathit{Code}\ \iota\ \nu) \\
\mid {} & \forall \iota'.\ \mathit{I}_X\ \ (\iota' \to \iota)\ \ (\mathit{Code}\ \iota'\ o) \\
\mid {} & \forall o'.\ \mathit{O}_X\ (o \to o')\ (\mathit{Code}\ \iota\ o') \\
\mid {} & \mu\ (\mathit{Code}\ (\mathit{Sum}\ \iota\ o)\ o)
\end{aligned}
$$

$$
\textbf{data}\ \mathit{Sum}\ \alpha\ \beta = \mathit{L}\ \alpha \mid \mathit{R}\ \beta
$$

Interpretation

$$
\begin{aligned}
&\textbf{data}\ (\llbracket\ \gamma :: \mathit{Code}\ \iota\ o\ \rrbracket) :: (\iota \to \star) \to (o \to \star)\ \textbf{where} \\
&\quad U' \ :: \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \llbracket\ U\ \rrbracket \qquad\quad \tau\ o \\
&\quad L' \ :: \llbracket\ \alpha\ \rrbracket\ \tau\ o \qquad\qquad\qquad\qquad \to \llbracket\ \alpha :+: \beta\ \rrbracket \quad \tau\ o \\
&\quad R' \ :: \llbracket\ \beta\ \rrbracket\ \tau\ o \qquad\qquad\qquad\qquad \to \llbracket\ \alpha :+: \beta\ \rrbracket \quad \tau\ o \\
&\quad \times' \ :: \llbracket\ \alpha\ \rrbracket\ \tau\ o \to \llbracket\ \beta\ \rrbracket\ \tau\ o \qquad \to \llbracket\ \alpha :\times: \beta\ \rrbracket \quad \tau\ o \\
&\quad C' \ :: \llbracket\ \alpha\ \rrbracket\ (\llbracket\ \beta\ \rrbracket\ \tau)\ o \qquad\qquad \to \llbracket\ \alpha :\cdot: \beta\ \rrbracket \quad \tau\ o \\
&\quad I' \ :: \tau\ \iota \qquad\qquad\qquad\qquad\qquad\qquad\quad \to \llbracket\ (I\ \iota)\ \rrbracket \qquad\quad \tau\ o \\
&\quad !' \ :: \llbracket\ \alpha\ \rrbracket\ \tau\ o \qquad\qquad\qquad\qquad\quad \to \llbracket\ (!\ \alpha\ o)\ \rrbracket \quad \tau\ o \\
&\quad I'_X \ :: \llbracket\ \alpha\ \rrbracket\ (\mathit{FComp}\ \tau\ \psi)\ o \qquad\quad \to \llbracket\ (I_X \quad \psi\ \alpha)\ \rrbracket\ \tau\ o \\
&\quad O'_X :: \llbracket\ \alpha\ \rrbracket\ \tau\ (\phi\ o) \qquad\qquad\quad \to \llbracket\ (O_X\ \phi\ \ \alpha)\ \rrbracket\ \tau\ o \\
&\quad \mu' \ :: \llbracket\ \phi\ \rrbracket\ (\mathit{Sum}_1\ \tau\ (\llbracket\ (\mu\ \phi)\ \rrbracket\ \tau))\ o \to \llbracket\ (\mu\ \phi)\ \rrbracket \qquad \tau\ o
\end{aligned}
$$

$$
\textbf{newtype}\ \mathit{FComp}\ \phi\ \psi\ \alpha = \mathit{FComp}\ \{\mathit{unFComp} :: \phi\ (\psi\ \alpha)\}
$$

$$
\begin{aligned}
&\textbf{data}\ \mathit{Sum}_1\ (\alpha :: \kappa_1 \to \star)\ (\beta :: \kappa_2 \to \star) :: \mathit{Sum}\ \kappa_1\ \kappa_2 \to \star\ \textbf{where} \\
&\quad L_1 :: \alpha\ \iota \to \mathit{Sum}_1\ \alpha\ \beta\ (L\ \iota) \\
&\quad R_1 :: \beta\ \iota \to \mathit{Sum}_1\ \alpha\ \beta\ (R\ \iota)
\end{aligned}
$$

Encoding lists

$$\begin{aligned}\textbf{type } List_F = (&U :+: ((I\ (L\ ())) :\times: (I\ (R\ ()))) \\ &:: Code\ (Sum\ ()\ ())\ ())\\ \textbf{type } List_C = (&\mu\ List_F :: Code\ ()\ ())\end{aligned}$$

Encoding lists

$$\begin{aligned}
&\textbf{type } \mathit{List}_F = (U :+: ((I\ (L\ ())) :\times: (I\ (R\ ())))\\
&\qquad\qquad :: \mathit{Code}\ (\mathit{Sum}\ ()\ ())\ ())\\
&\textbf{type } \mathit{List}_C = (\mu\ \mathit{List}_F :: \mathit{Code}\ ()\ ())
\end{aligned}$$

$$\textbf{data } \mathit{Const}\ \alpha\ \beta = \mathit{Const}\ \{\mathit{unConst} :: \alpha\}$$

$$\begin{aligned}
&\mathit{fromList} :: [\alpha] \to [\![\ \mathit{List}_C\ ]\!]\ (\mathit{Const}\ \alpha)\ ()\\
&\mathit{fromList}\ [] = \mu'\ (L'\ U')\\
&\mathit{fromList}\ (x : xs) = \mu'\ (R'\ (\times'\ (I'\ (L_1\ (\mathit{Const}\ x)))\ (I'\ (R_1\ (\mathit{fromList}\ xs)))))
\end{aligned}$$

$$\begin{aligned}
&\mathit{toList} :: [\![\ \mathit{List}_C\ ]\!]\ (\mathit{Const}\ \alpha)\ () \to [\alpha]\\
&\mathit{toList}\ (\mu'\ (L'\ U')) = []\\
&\mathit{toList}\ (\mu'\ (R'\ (\times'\ (I'\ (L_1\ (\mathit{Const}\ x)))\ (I'\ (R_1\ xs))))) = x : \mathit{toList}\ xs
\end{aligned}$$

Encoding rose trees

Rose trees use lists:

$$\textbf{data}\ \mathit{Rose}\ \alpha = \mathit{Fork}\ \alpha\ [\mathit{Rose}\ \alpha]$$

Encoding rose trees

Rose trees use lists:

$$\textbf{data}\ \mathit{Rose}\ \alpha = \mathit{Fork}\ \alpha\ [\mathit{Rose}\ \alpha]$$

So we reuse the encoding for lists, with composition:

$$\begin{aligned}
&\textbf{type}\ \mathit{Rose}_F = ((I\ (L\ ()))\ :\times:\ (\mathit{List}_C\ :\cdot:\ (I\ (R\ ())))\\
&\qquad\qquad :: \mathit{Code}\ (\mathit{Sum}\ ()\ ())\ ())\\
&\textbf{type}\ \mathit{Rose}_C = (\mu\ \mathit{Rose}_F :: \mathit{Code}\ ()\ ())
\end{aligned}$$

$$\begin{aligned}
&\mathit{fromRose} :: \mathit{Rose}\ \alpha \to [\![\ \mathit{Rose}_C\ ]\!]\ (\mathit{Const}\ \alpha)\ ()\\
&\mathit{fromRose}\ (\mathit{Fork}\ a\ as) =\\
&\quad \mu'\ (\times'\ (I'\ (L_1\ (\mathit{Const}\ a)))\ (C'\\
&\qquad (\mathit{imap}\ (I' \circ R_1 \circ \mathit{fromRose} \circ \mathit{unConst})\ (\mathit{fromList}\ as))))
\end{aligned}$$

$$\begin{aligned}
&\mathit{toRose} :: [\![\ \mathit{Rose}_C\ ]\!]\ (\mathit{Const}\ \alpha)\ () \to \mathit{Rose}\ \alpha\\
&\mathit{toRose}\ (\mu'\ (\times'\ (I'\ (L_1\ (\mathit{Const}\ a)))\ (C'\ as))) =\\
&\quad \mathit{Fork}\ a\ (\mathit{toList}\ (\mathit{imap}\ (\mathit{Const} \circ \mathit{toRose} \circ \mathit{unR}_1 \circ \mathit{un}[\![]\!]_I)\ as))
\end{aligned}$$

Reindexing I

Consider the following example:

$$\begin{aligned}
&\textbf{data}\ \mathit{Expr}\ \alpha = \mathit{Var}\ \alpha \\
&\qquad\qquad\quad \mid \mathit{Let}\ (\mathit{Decl}\ \alpha)\ (\mathit{Expr}\ \alpha) \\
&\textbf{data}\ \mathit{Decl}\ \alpha = \mathit{Assign}\ \alpha\ (\mathit{Expr}\ \alpha) \\
&\qquad\qquad\quad \mid \mathit{Seq}\ [\mathit{Decl}\ \alpha]
\end{aligned}$$

Reindexing I

Consider the following example:

$$\begin{aligned}
&\textbf{data}\ \mathit{Expr}\ \alpha = \mathit{Var}\ \alpha \\
&\qquad\qquad\quad \mid \mathit{Let}\ (\mathit{Decl}\ \alpha)\ (\mathit{Expr}\ \alpha) \\
&\textbf{data}\ \mathit{Decl}\ \alpha = \mathit{Assign}\ \alpha\ (\mathit{Expr}\ \alpha) \\
&\qquad\qquad\quad \mid \mathit{Seq}\ [\mathit{Decl}\ \alpha]
\end{aligned}$$

Given the kind of composition:

$$(:\cdot:) :: \forall \nu_\kappa.\ \mathit{Code}\ \nu_\kappa\ o_\kappa \to \mathit{Code}\ \iota_\kappa\ \nu_\kappa \to \mathit{Code}\ \iota_\kappa\ o_\kappa$$

Reindexing I

Consider the following example:

$$\begin{aligned}&\textbf{data}\ \mathit{Expr}\ \alpha = \mathit{Var}\ \alpha\\ &\qquad\qquad\mid \mathit{Let}\ (\mathit{Decl}\ \alpha)\ (\mathit{Expr}\ \alpha)\\ &\textbf{data}\ \mathit{Decl}\ \alpha = \mathit{Assign}\ \alpha\ (\mathit{Expr}\ \alpha)\\ &\qquad\qquad\mid \mathit{Seq}\ [\mathit{Decl}\ \alpha]\end{aligned}$$

Given the kind of composition:

$$(:\cdot:) :: \forall \nu_\kappa.\ \mathit{Code}\ \nu_\kappa\ o_\kappa \to \mathit{Code}\ \iota_\kappa\ \nu_\kappa \to \mathit{Code}\ \iota_\kappa\ o_\kappa$$

To encode *Decl* we need to use reindexing:

$$\begin{aligned}&\textbf{data}\ \mathit{AST}_I = \mathit{Expr}_I \mid \mathit{Decl}_I\\ &\textbf{type}\ \mathit{List}_{\uparrow \mathit{AST}} = (O_X\ \mathit{List}_{\uparrow \mathit{AST}_O}\ \mathit{List}_C :: \mathit{Code}\ ()\ \mathit{AST}_I)\\ &\textbf{type}\ \mathit{Decl}_F = (((I\ (L\ ()))\ :\times:\ (I\ (R\ \mathit{Expr}_I)))\ :+:\ (\mathit{List}_{\uparrow \mathit{AST}}\ :\cdot:\ (I\ (R\ \mathit{Expr}_I)))\\ &\qquad\qquad :: \mathit{Code}\ (\mathit{Sum}\ ()\ \mathit{AST}_I)\ \mathit{AST}_I)\end{aligned}$$

Reindexing II

The reindexing is performed by a type family:

$$\textbf{type family } List_{\uparrow AST_O} :: AST_I \to ()$$

Unfortunately we cannot give any meaningful instances for this type family. To be able to match on the indices on the left we would need to declare it as follows:

$$\textbf{type family } List_{\uparrow AST_O}\ (o :: AST_I) :: ()$$

But GHC requires type family applications to be fully saturated...

Conclusion and future work

- The new features are great for generic programming

Conclusion and future work

- The new features are great for generic programming
- . . . but we want more!

Conclusion and future work

- The new features are great for generic programming
- . . . but we want more!
  - Defining a kind without defining a datatype

Conclusion and future work

- The new features are great for generic programming
- . . . but we want more!
  - Defining a kind without defining a datatype
  - Kind synonyms

Conclusion and future work

- The new features are great for generic programming
- . . . but we want more!
  - Defining a kind without defining a datatype
  - Kind synonyms
  - Solvers for type-level naturals and symbols

Conclusion and future work

- The new features are great for generic programming
- . . . but we want more!
  - Defining a kind without defining a datatype
  - Kind synonyms
  - Solvers for type-level naturals and symbols
  - Exhaustiveness checking for instances on a promoted datatype

Conclusion and future work

- The new features are great for generic programming
- . . . but we want more!
  - Defining a kind without defining a datatype
  - Kind synonyms
  - Solvers for type-level naturals and symbols
  - Exhaustiveness checking for instances on a promoted datatype
  - Promoted GADTs

Conclusion and future work

- The new features are great for generic programming
- . . . but we want more!
  - Defining a kind without defining a datatype
  - Kind synonyms
  - Solvers for type-level naturals and symbols
  - Exhaustiveness checking for instances on a promoted datatype
  - Promoted GADTs
  - Unsaturated type family applications (useful with promoted functions)

Conclusion and future work

- The new features are great for generic programming
- . . . but we want more!
  - Defining a kind without defining a datatype
  - Kind synonyms
  - Solvers for type-level naturals and symbols
  - Exhaustiveness checking for instances on a promoted datatype
  - Promoted GADTs
  - Unsaturated type family applications (useful with promoted functions)

Open questions:

- Can we remove the duplication in $liftM/liftM_2$ and $zipWith/zipWith_3$?

Conclusion and future work

- The new features are great for generic programming
- . . . but we want more!
  - Defining a kind without defining a datatype
  - Kind synonyms
  - Solvers for type-level naturals and symbols
  - Exhaustiveness checking for instances on a promoted datatype
  - Promoted GADTs
  - Unsaturated type family applications (useful with promoted functions)

Open questions:

- Can we remove the duplication in $liftM/liftM_2$ and $zipWith/zipWith_3$?
- How to encode the reindexing operation?

Conclusion and future work

- The new features are great for generic programming
- . . . but we want more!
  - Defining a kind without defining a datatype
  - Kind synonyms
  - Solvers for type-level naturals and symbols
  - Exhaustiveness checking for instances on a promoted datatype
  - Promoted GADTs
  - Unsaturated type family applications (useful with promoted functions)

Open questions:

- Can we remove the duplication in $liftM/liftM_2$ and $zipWith/zipWith_3$?
- How to encode the reindexing operation?
- Is performance affected by the use of kinds?