Giving Haskell a Promotion

José Pedro Magalhães

Joint work with
Brent A. Yorgey  Stephanie Weirich  Julien Cretin
Simon Peyton Jones  Dimitrios Vytiniotis

http://www.dreixel.net

FP dag 2012, Universiteit Utrecht, 06/01/2012
This talk is about the new -XPolyKinds flag in GHC 7.4. This flag promotes two well known type features of Haskell to the kind level: algebraic datatypes and polymorphism. This allows us to write more precise types in our programs, and to reduce some code duplication.
Colors

In this talk I will use:

- Blue for constructors (most of the time)
  *Nothing, False, Left True, 3, "abc", ’p’*

- Red for types
  *Int, String, Show a data () = ()*

- Green for kinds
  *⋆, ⋆ → ⋆*
Colors

In this talk I will use:

- Blue for constructors (most of the time)
  *Nothing*, *False*, *Left True*, 3, "abc", ’p’

- Red for types
  *Int*, *String*, *Show a*, \texttt{data () = ()}
Colors

In this talk I will use:

- Blue for constructors (most of the time)
  \[ \text{Nothing}, \text{False}, \text{Left True}, 3, "abc", 'p' \]
- Red for types
  \[ \text{Int}, \text{String}, \text{Show a}, \text{data () = ()} \]
- Green for kinds
  \[ \star, \star \to \star \]
What are kinds?

Just like types classify values...

\[
3 :: \text{Num } a \Rightarrow a \\
'p' :: \text{Char} \\
Just () :: \text{Maybe } () \\
"abc" :: \text{String}
\]
What are kinds?

Just like types classify values...

3 :: Num a ⇒ a
’p’ :: Char
Just () :: Maybe ()
"abc" :: String

... kinds classify types:

Int :: ⊤
Char :: ⊤
Maybe :: ⊤ → ⊤
[] :: ⊤ → ⊤
The language of kinds

However, the language of kinds, unlike that of types, is rather limited:

\[ k ::= \star \]
\[ \mid k \to k \]

In particular: no user defined kinds, no kind variables.
(Caveat: we are ignoring \# and friends for this talk.)
Diversion: the *Constraint* kind

With \(-X\)ConstraintKinds we get one new base kind to classify constraints:

\[
\begin{align*}
\text{Show} & \quad :: \star \to Constraint \\
\text{Functor} & \quad :: (\star \to \star) \to Constraint \\
\text{Num Int} & \quad :: Constraint \\
\text{Int} \sim \text{Bool} & \quad :: Constraint
\end{align*}
\]
Why do we need a better kind system? I

We often want to restrict type arguments to a particular kind:

```
data Ze
data Su n

data Vec :: ⋆ → ⋆ → ⋆ where
    Nil  :: Vec a Ze
    Cons :: a → Vec a n → Vec a (Su n)
```

Types like `Vec Int Int`, `Vec Int Bool`, and `Vec () ()` are valid (albeit uninhabited). We want to say that the second argument of `Vec` should only be `Ze` or `Su`!
Lack of kind polymorphism leads to code duplication:

```haskell
class Typeable (a :: ★) where
typeOf :: a → TypeRep

class Typeable_1 (a :: ★ → ★) where
typeOf_1 :: a b → TypeRep

class Typeable_2 (a :: ★ → ★ → ★) where
typeOf_2 :: a b c → TypeRep
```

We would rather have a single, kind-polymorphic `Typeable` class!
Datatype promotion I

With -XPolyKinds, the following code is valid:

```haskell
data Nat = Ze | Su Nat

data Vec :: ⋆ → Nat → ⋆ where
    Nil   :: Vec a Ze
    Cons :: a → Vec a n → Vec a (Su n)
```

Note the implicit promotion of the constructors Ze and Su to types Ze and Su, and of the type Nat to the kind Nat.

Types like Vec Int Int now trigger a kind error!
Datatype promotion II

Only non-indexed datatypes with parameters of kind $\star$ can be promoted. So the following are ok:

\begin{array}{ll}
data \ Bool &= \ True \mid False \\
data \ Tree \ a &= \ Leaf \mid Bin \ a \ (Tree \ a) \ (Tree \ a) \\
data \ Rose \ a &= \ Rose \ a \ [Rose \ a] \\
data \ Perfect \ a &= \ Split \ (Perfect \ (a, a)) \mid Element \ a \\
\end{array}
Datatype promotion II

Only non-indexed datatypes with parameters of kind \( \star \) can be promoted. So the following are ok:

```haskell
data Bool = True | False
data Tree a = Leaf | Bin a (Tree a) (Tree a)
data Rose a = Rose a [Rose a]
data Perfect a = Split (Perfect (a, a)) | Element a
```

But the following are not promoted:

```haskell
data Fix f = In (f (Fix f))
data Dynamic = \( \forall t. \text{Typeable } t \Rightarrow \text{Dyn } t \)
data Vec :: \( \star \rightarrow \text{Nat} \rightarrow \star \) where
  Nil :: Vec a Ze
  Cons :: a \rightarrow \text{Vec } a \ n \rightarrow \text{Vec } a \ (\text{Su } n)
```
Datatype promotion III

Type families can also be indexed over promoted types:

```haskell
type family Add (m :: Nat) (n :: Nat) :: Nat

type instance Add Ze n = n

type instance Add (Sum m) n = Sum (Add m n)
```

This was all possible before, but now we can express the right kind of `Add`.
Datatype promotion III

Type families can also be indexed over promoted types:

\[
\text{type family } \text{Add} \ (m :: \text{Nat}) \ (n :: \text{Nat}) :: \text{Nat}
\]

\[
\text{type instance } \text{Add} \ \text{Ze} \quad n = n
\]

\[
\text{type instance } \text{Add} \ (\text{Su} \ m) \ n = \text{Su} \ (\text{Add} \ m \ n)
\]

\[
\text{append} :: \text{Vec} \ a \ m \rightarrow \text{Vec} \ a \ n \rightarrow \text{Vec} \ a \ (\text{Add} \ m \ n)
\]

\[
\text{append } \text{Nil} \quad v = v
\]

\[
\text{append } (\text{Cons} \ h \ t) \ v = \text{Cons} \ h \ (\text{append} \ t \ v)
\]

This was all possible before, but now we can express the right kind of \textit{Add}.  

Promoted lists and tuples

Haskell lists are natively promoted, so we can encode heterogeneous lists as follows:

```haskell
data HList :: [⋆] → ⋆ where
  HNil    :: HList []
  HCons   :: a → HList t → HList (a : t)
```

As an example, here is a heterogeneous collection:

```haskell
hetList :: HList [Int, Bool]
hetList = HCons 3 (HCons False HNil)
```
Promoted lists and tuples

Haskell lists are natively promoted, so we can encode heterogeneous lists as follows:

\[
\text{data } \text{HList} :: \star \rightarrow \star \text{ where}
\]
\[
\text{HNil} :: \text{HList } []
\]
\[
\text{HCons} :: a \rightarrow \text{HList } t \rightarrow \text{HList } (a : t)
\]

As an example, here is a heterogeneous collection:

\[
\text{hetList} :: \text{HList } [\text{Int}, \text{Bool}]
\]
\[
\text{hetList} = \text{HCons } 3 (\text{HCons } \text{False } \text{HNil})
\]

Tuples are also promoted, e.g. \((\star, \star \rightarrow \star, \text{Constraint})\).
Kind-polymorphic type equality

Kind polymorphism reduces code duplication:

\[
\text{data } Eq_T \ a \ b \ \text{where} \\
\quad \text{Refl} :: Eq_T \ a \ a
\]

Previously the kind of \( Eq_T \) would default to \( \star \to \star \to \star \). With kind polymorphism it doesn’t, so the following types are all valid: \( Eq_T \ a \ \text{Int} \), \( Eq_T \ f \ \text{Maybe} \), \( Eq_T \ t \ \text{Either} \).
Now we can define a single kind-polymorphic `Typeable` class:

```haskell
data Proxy t = Proxy

class Typeable (t :: k) where
  typeRep :: Proxy t → TypeRep
```

Note that `Proxy` is kind polymorphic!
Kind-polymorphic Typeable II

We can give Typeable instances for types of various kinds:

instance Typeable Char where...
instance Typeable [] where...
instance Typeable Either where...
For backwards compatibility, the old methods can be defined by instantiating `typeRep` to the right kind:

\[
\text{typeOf :: } \forall a. \text{Typeable } a \Rightarrow a \rightarrow \text{TypeRep} \\
\text{typeOf } x = \text{typeRep (getType } x) \text{ where} \\
\text{getType :: } a \rightarrow \text{Proxy } a \\
\text{getType } _\_ = \text{Proxy}
\]
Kind-polymorphic \textit{Typeable III}

For backwards compatibility, the old methods can be defined by instantiating \textit{typeRep} to the right kind:

\[
\begin{align*}
\text{typeOf} & :: \forall a. \text{Typeable } a \Rightarrow a \to \text{TypeRep} \\
\text{typeOf} \ x &= \text{typeRep} \ (\text{getType} \ x) \ \text{where} \\
\text{getType} & :: a \to \text{Proxy } a \\
\text{getType} \ _ &= \text{Proxy}
\end{align*}
\]

\[
\begin{align*}
\text{typeOf}_1 & :: \forall f \ (a :: \star). \text{Typeable } f \Rightarrow f \ a \to \text{TypeRep} \\
\text{typeOf}_1 \ x &= \text{typeRep} \ (\text{getType}_1 \ x) \ \text{where} \\
\text{getType}_1 & :: f \ a \to \text{Proxy } f \\
\text{getType}_1 \ _ &= \text{Proxy}
\end{align*}
\]
Kind-polymorphic fixed-point operator

A single fixed-point operator for types of different kinds:

\[
\text{data } Mu f a = Roll (f (Mu f) a)
\]
Kind-polymorphic fixed-point operator

A single fixed-point operator for types of different kinds:

\[
\text{data } \textit{Mu } f \ a = \textit{Roll} \ (f \ (\textit{Mu } f) \ a)
\]

Here \( \textit{Mu} \) is instantiated to kind \(((\star \rightarrow \star) \rightarrow \star \rightarrow \star) \rightarrow \star \rightarrow \star\):

\[
\text{data } \textit{ListF } f \ a = \textit{NilF} \mid \textit{ConsF } a \ (f \ a)
\]

\text{type } \textit{List} \ a = \textit{Mu } \textit{ListF } a
Kind-polymorphic fixed-point operator

A single fixed-point operator for types of different kinds:

```haskell
data Mu f a = Roll (f (Mu f) a)
```

Here `Mu` is instantiated to kind `((\* \to \* ) \to \* \to \* ) \to \* \to \* )`:

```haskell
data ListF f a = NilF | ConsF a (f a)
type List a = Mu ListF a
```

And here to kind `((Nat \to \*) \to Nat \to \*) \to Nat \to \* )`:

```haskell
data VecF (a :: \*) (f :: Nat \to \*) (n :: Nat) where
  VFNil :: VecF a f Ze
  VFCons :: a \to f n \to VecF a f (Su n)
type Vec a n = Mu (VecF a) n
```
Better kinded generics: representation

Generic programming becomes clearer because we can set the representation types apart in a separate kind.
Better kinded generics: representation

Generic programming becomes clearer because we can set the representation types apart in a separate kind.

```
data Universe x = U |
                 K x |
                 Universe x ⊕ Universe x |
                 Universe x × Universe x
```
Better kinded generics: representation

Generic programming becomes clearer because we can set the representation types apart in a separate kind.

```
data Universe x = U
    | K x
    | Universe x ∪ Universe x
    | Universe x × Universe x
```

```
data Interprt :: Universe ⋆ → ⋆ where

U :: Interprt U
K :: t → Interprt (K t)
L :: Interprt a → Interprt (a + b)
R :: Interprt b → Interprt (a + b)
(×) :: Interprt a → Interprt b → Interprt (a × b)
```
Better kinded generics: conversion

Class mediating the conversion between user types and their generic representation:

```haskell
class Representable a where
    type Rep a :: Universe ⋆
    from :: a → Interprt (Rep a)
    to :: Interprt (Rep a) → a
```
Better kinded generics: conversion

Class mediating the conversion between user types and their generic representation:

class Representable a where
    type Rep a :: Universe ⋆
    from :: a → Interprt (Rep a)
    to     :: Interprt (Rep a) → a

User types are (automatically) representable:

instance Representable [a] where...
instance Representable (Maybe a) where...
instance Representable (Either a b) where...
Better kinded generics: functions

User-visible class, exported:

class Show (a :: *) where
    show :: a → String

default show :: (Representable a, GShow (Rep a)) ⇒ a → String
    show = gshow ∘ from
Better kinded generics: functions

User-visible class, exported:

```haskell
class Show (a :: *) where
    show :: a → String

default show :: (Representable a, GShow (Rep a)) → a → String
    show = gshow ◦ from
```

Defined by the generic programmer, not exported:

```haskell
class GShow (a :: Universe *) where
    gshow :: Interprt a → String

instance GShow U where
    gshow U = ""
    ...
```
Summary

This new GHC extension:

- Increases type safety
Summary

This new GHC extension:

▶ Increases type safety
▶ Increases expressivity
Summary

This new GHC extension:

▶ Increases type safety
▶ Increases expressivity
▶ Reduces code duplication
▶ Allows for writing clearer code
▶ Brings us one step closer to dependently-typed programming in Haskell
Summary

This new GHC extension:

- Increases type safety
- Increases expressivity
- Reduces code duplication
- Allows for writing clearer code
Summary

This new GHC extension:

- Increases type safety
- Increases expressivity
- Reduces code duplication
- Allows for writing clearer code
- Brings us one step closer to dependently-typed programming in Haskell
Future work

On the pipeline:

▶ Explicit kind variable annotations
Future work

On the pipeline:

- Explicit kind variable annotations
- Kind synonyms (from type synonym promotion)
Future work

On the pipeline:

- Explicit kind variable annotations
- Kind synonyms (from type synonym promotion)
- Template Haskell support
Future work

On the pipeline:

► Explicit kind variable annotations
► Kind synonyms (from type synonym promotion)
► Template Haskell support

To think about:

► Generalized Algebraic Data Kinds