GHC 7.6, More Well-Typed Than Ever

José Pedro Magalhães

http://www.dreixel.net

DHD ➞ UHac 2012, Universiteit Utrecht, 20/04/2012
This talk is about upcoming and exciting features in GHC 7.6:

- Data kinds
- Kind polymorphism
- Type-level literals
- Deferred type errors

We’ll go through a few examples of how to put these new features to good use.

(Note that this is all still work in progress, and implementation details might change!)
Colours

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`, `data () = ()`

- Green for kinds
  - `⋆`, `⋆ → ⋆`
Colours

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 () = ()
Colours

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
  \(\star, \star \rightarrow \star\)
1. Kinds
What are kinds?

Just like types classify values...

\[3 :: \text{Num } a \Rightarrow a\]

\['p' :: \text{Char}\]

\[\text{Just () :: Maybe ()}\]

\"abc" :: \text{String}\]
What are kinds?

Just like types classify values...

\[ 3 :: \text{Num} \ a \Rightarrow a \]

\[ 'p' :: \text{Char} \]

\[ \text{Just} () :: \text{Maybe} () \]

\[ "abc" :: \text{String} \]

... kinds classify types:

\[ \text{Int} :: \ast \]

\[ \text{Char} :: \ast \]

\[ \text{Maybe} :: \ast \rightarrow \ast \]

\[ [] :: \ast \rightarrow \ast \]
The language of kinds

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

\[ k ::= \star \]
\[ | \quad k \rightarrow k \]

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

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

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

Lack of kind polymorphism leads to code duplication:

\[
\begin{align*}
\textbf{class } & \textsf{Typeable} \ (a :: \star) \quad \text{where} \\
& \quad \textit{typeOf} :: a \to \textit{TypeRep} \\
\textbf{class } & \textsf{Typeable}_1 \ (a :: \star \to \star) \quad \text{where} \\
& \quad \textit{typeOf}_1 :: a \ b \to \textit{TypeRep} \\
\textbf{class } & \textsf{Typeable}_2 \ (a :: \star \to \star \to \star) \quad \text{where} \\
& \quad \textit{typeOf}_2 :: a \ b \ c \to \textit{TypeRep}
\end{align*}
\]

We would rather have a single, kind-polymorphic \textit{Typeable} class!
Datatype promotion I

With {-XDataKinds}, 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:

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

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

\begin{align*}
data & \text{Bool} & = & \text{True} \mid \text{False} \\
data & \text{Tree } a & = & \text{Leaf} \mid \text{Bin } a (\text{Tree } a) (\text{Tree } a) \\
data & \text{Rose } a & = & \text{Rose } a \ [ \text{Rose } a] \\
data & \text{Perfect } a & = & \text{Split} (\text{Perfect} (a, a)) \mid \text{Element } a
\end{align*}

But the following are not promoted:

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

Type families can also be indexed over promoted types:

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

type instance Add Z e n = n

type instance Add (S u m) n = S u (Add m 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)
```

```haskell
append :: Vec a m -> Vec a n -> Vec a (Add m n)
append Nil v = v
append (Cons h t) v = Cons h (append t v)
```

This was all possible before, but now we can express the right kind of `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)
```

Tuples are also promoted, e.g. \((⋆, ⋆ → ⋆, Constraint)\).
Kind-polymorphic type equality

Kind polymorphism reduces code duplication:

```haskell
data Eq_T a b where
  Refl :: Eq_T a a
```

Previously the kind of $Eq_T$ would default to $\star \rightarrow \star \rightarrow \star$. With `-XPolyKinds` 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 \textit{Typeable} class:

\begin{verbatim}
data Proxy (t :: k) = Proxy
class Typeable (t :: k) where
typeRep :: Proxy t \rightarrow TypeRep
\end{verbatim}

Note that \textit{Proxy} is kind polymorphic!
We can give *Typeable* instances for types of various kinds:

```haskell
instance Typeable Char where...
instance Typeable [] where...
instance Typeable Either where...
```
Kind-polymorphic *Typeable* III

For backwards compatibility, the old methods can be defined by instantiating `typeRep` to the right kind:

\[
\text{typeof} :: \forall a. \text{Typeable } a \Rightarrow a \rightarrow \text{TypeRep} \\
\text{typeof } x = \text{typeRep} (\text{getType } x) \quad \text{where} \\
\text{getType} :: a \rightarrow \text{Proxy } a \\
\text{getType } _ = \text{Proxy}
\]
For backwards compatibility, the old methods can be defined by instantiating `typeRep` to the right kind:

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

\[
\text{typeOf}_1 :: \forall f \ (a :: \star). \text{Typeable } f \Rightarrow f \ a \rightarrow \text{TypeRep} \\
\text{typeOf}_1 x = \text{typeRep } (\text{getType}_1 x) \text{ where} \\
\text{getType}_1 :: f \ a \rightarrow \text{Proxy } f \\
\text{getType}_1 _{-} = \text{Proxy}
\]
2. Type-level literals
Type-level literals

Thanks to Iavor Diatchki’s hard work, we will have efficient type-level naturals:

\[ 0, 1, 2, \ldots :: \text{Nat} \]

Note the colours!

These type-level naturals come with associated operations:

\[
\begin{align*}
(\leq) & :: \text{Nat} \to \text{Nat} \to \text{Constraint} \\
(+) & :: \text{Nat} \to \text{Nat} \to \text{Nat} \\
(*) & :: \text{Nat} \to \text{Nat} \to \text{Nat} \\
(^) & :: \text{Nat} \to \text{Nat} \to \text{Nat}
\end{align*}
\]
Value-level reflection

How do we manipulate values representing type-level naturals? There is a family of singleton types, parameterised by literals:

\[
\text{newtype } Sing :: a \rightarrow * \\
\]

Note that we can have type-level literals other than naturals, and \( \text{SingRep} \) is a kind-indexed family!
Value-level reflection

How do we manipulate values representing type-level naturals? There is a family of singleton types, parameterised by literals:

\[
\text{newtype } \text{Sing} :: a \rightarrow \star
\]

From types to values:

\[
\text{fromSing} :: \text{Sing } a \rightarrow \text{SingRep } a
\]

\[
\text{type family } \text{SingRep } a
\]

\[
\text{type instance } \text{SingRep } (a :: \text{Nat}) = \text{Integer}
\]

\[
\text{type instance } \text{SingRep } (a :: \text{Symbol}) = \text{String}
\]

Note that we can have type-level literals other than naturals, and \text{SingRep} is a kind-indexed family!
Revisiting vectors, now with type-level naturals:

```haskell
data Vec :: Nat → ⋆ → ⋆ where
  Nil    :: Vec 0 a
  Cons :: a → Vec n a → Vec (n + 1) a
```

Vector concatenation uses type-level natural number addition:

```haskell
append :: Vec m a → Vec n a → Vec (m + n) a
append Nil ys = ys
append (Cons x xs) ys = Cons x (append xs ys)
```
Why are type-level naturals hard to implement?

Function *append* requires GHC to prove equalities between natural number expressions:

- Could not deduce \((n \sim (0 + n))\) from the context \((m \sim 0)\) bound by a pattern with constructor \(\text{Nil} :: \forall a. \text{Vec} \ 0 \ a\)

- Could not deduce \(((m + n) \sim ((n' + n) + 1))\) from the context \((m \sim (n' + 1))\) bound by a pattern with constructor

\[\text{Cons} :: \forall a \ (n :: \text{Nat}). a \rightarrow \text{Vec} \ n \ a \rightarrow \text{Vec} \ (n + 1) \ a\]

We need an equation solver!
3. Deferring type errors
The illogical next step

What is the next thing that you want, when you have data kinds, polymorphic kinds, and type-level literals?

Naturally, to turn off type checking! :-(
The illogical next step

What is the next thing that you want, when you have data kinds, polymorphic kinds, and type-level literals?

Naturally, to turn off type checking! :-}
Why would you want to do that?

For instance:

▶ Prototyping
▶ Large refactoring
▶ IDE
Example 1

With the flag `-fdefer-type-errors`, this example:

\[
\begin{align*}
p, q &: \text{Int} \\
p &= 1 \\
q &= '1' \\
main &= \text{print } p
\end{align*}
\]

Compiles with warning: “couldn’t match expected type \text{Int} with actual type \text{Char} in an equation for \( q: q = '1' \)”. Runs and returns 1.
Example II

\[ p, q :: \text{Int} \]
\[ p = 1 \]
\[ q = '1' \]
\[ \text{main} = \text{print } q \]

Fails at runtime with: “couldn’t match expected type \text{Int} with actual type \text{Char} in an equation for \( q: q = '1' \)."
Example III

\[ t_1 :: \text{Int} \]
\[ t_1 = '1' \]
\[ t_2 :: a \rightarrow \text{String} \]
\[ t_2 = \text{show} \]

\textbf{data} \ T \ a \ \textbf{where} \\
\hspace{1em} T_1 :: \text{Int} \rightarrow T \ \text{Int} \\
\hspace{1em} T_2 :: a \rightarrow T \ a \\
\[ t_3 :: T \ a \]
\[ t_3 = T_1 0 \]

\textit{main} = \text{print} 1

\textbf{Runs fine!}
How it works

GHC’s core language uses coercions to (safely) cast terms:

```haskell
data T a = T1 (a ~ Int) Int | T2 a

unT :: T a → a

unT (T1 c n) = n ⊢ (sym c)

unT (T2 x) = x

⊢ :: b → (b ~ a) → a
```

Evidence, or values of type (~), is automatically generated by GHC during type checking. Deferring type errors simply means generating runtime errors as evidence!

(The complete story is a bit more involved; see the paper for details!)
It’s not dynamic typing!

Note that deferring type errors doesn’t mean any form of checks are performed at runtime. Consider this example:

\[
f :: \forall a. a \to a \to a \\
f x y = x \land y \\
main = \text{print} (f \ True \ False)
\]

It still fails at runtime!
Summary

A better kind system gives us:

- Increase type safety
- Increase expressivity
- Reduce code duplication
- Allow for writing clearer code

And if we get tired of it we can always defer errors to runtime!
Summary

A better kind system gives us:

- Increase type safety
- Increase expressivity
- Reduce code duplication
- Allow for writing clearer code

And if we get tired of it we can always defer errors to runtime!
Future work

On the pipeline:

- Kind synonyms (from type synonym promotion)
- Template Haskell support
- A solver for type-level naturals
Future work

On the pipeline:

- Kind synonyms (from type synonym promotion)
- Template Haskell support
- A solver for type-level naturals

To think about:

- Generalized Algebraic Data Kinds
- User-defined solvers
- Deferring kind errors?