Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

GHC 7.6, More Well-Typed Than Ever

José Pedro Magalhães

http://www.dreixel.net

DHD ≫= UHac 2012, Universiteit Utrecht, 20/04/2012

Introduction

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'`

Colours

In this talk I will use:

- Blue for constructors (most of the time)
  $\mathit{Nothing}$, False, $\mathit{Left}\ \mathit{True}$, 3, `"abc"`, `'p'`
- Red for types
  Int, String, $\mathit{Show}\ a$, $\mathbf{data}\ () = ()$

Colours

In this talk I will use:

- Blue for constructors (most of the time)
  $\mathit{Nothing}$, False, $\mathit{Left}\ \mathit{True}$, 3, `"abc"`, `'p'`
- Red for types
  Int, String, $\mathit{Show}\ a$, $\mathbf{data}\ () = ()$
- Green for kinds
  $\star$, $\star \rightarrow \star$

1. Kinds

What are kinds?

Just like types classify values...

$$\begin{aligned} &3 && :: Num\ a \Rightarrow a \\ &\texttt{'p'} && :: Char \\ &Just\ () && :: Maybe\ () \\ &\texttt{"abc"} && :: String \end{aligned}$$

What are kinds?

Just like types classify values...

$$\begin{aligned}
3 &:: \mathit{Num}\ a \Rightarrow a \\
\texttt{'p'} &:: \mathit{Char} \\
\mathit{Just}\ () &:: \mathit{Maybe}\ () \\
\texttt{"abc"} &:: \mathit{String}
\end{aligned}$$

... kinds classify types:

$$\begin{aligned}
\mathit{Int} &:: \star \\
\mathit{Char} &:: \star \\
\mathit{Maybe} &:: \star \to \star \\
[\,] &:: \star \to \star
\end{aligned}$$

The language of kinds

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

$$\begin{aligned} k &::= \star \\ &\mid\ \ k \to k \end{aligned}$$

In particular: no user defined kinds, no kind variables.

(Caveat: we are ignoring $\#$ and friends for this talk.)

Diversion: the $\mathit{Constraint}$ kind

With `-XConstraintKinds` we get one new base kind to classify constraints:

$$
\begin{aligned}
&\mathit{Show} &&:: \star \to \mathit{Constraint} \\
&\mathit{Functor} &&:: (\star \to \star) \to \mathit{Constraint} \\
&\mathit{Num}\ \mathit{Int} &&:: \mathit{Constraint} \\
&\mathit{Int} \sim \mathit{Bool} &&:: \mathit{Constraint}
\end{aligned}
$$

Why do we need a better kind system? I

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

$$\begin{array}{l}\textbf{data } \mathit{Ze} \\ \textbf{data } \mathit{Su}\ n\end{array}$$

$$\begin{array}{l}\textbf{data } \mathit{Vec} :: \star \to \star \to \star \textbf{ where} \\ \quad \mathit{Nil} \quad :: \mathit{Vec}\ a\ \mathit{Ze} \\ \quad \mathit{Cons} :: a \to \mathit{Vec}\ a\ n \to \mathit{Vec}\ a\ (\mathit{Su}\ n)\end{array}$$

Types like $\mathit{Vec}\ \mathit{Int}\ \mathit{Int}$, $\mathit{Vec}\ \mathit{Int}\ \mathit{Bool}$, and $\mathit{Vec}\ ()\ ()$ 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:

$$
\begin{array}{l}
\textbf{class}\ \mathit{Typeable}\ (a :: \star)\ \textbf{where} \\
\quad \mathit{typeOf} :: a \to \mathit{TypeRep} \\
\textbf{class}\ \mathit{Typeable}_1\ (a :: \star \to \star)\ \textbf{where} \\
\quad \mathit{typeOf}_1 :: a\ b \to \mathit{TypeRep} \\
\textbf{class}\ \mathit{Typeable}_2\ (a :: \star \to \star \to \star)\ \textbf{where} \\
\quad \mathit{typeOf}_2 :: a\ b\ c \to \mathit{TypeRep}
\end{array}
$$

We would rather have a single, kind-polymorphic *Typeable* class!

Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

Datatype promotion I

With `-XDataKinds`, the following code is valid:

$$\mathbf{data}\ \mathit{Nat} = \mathit{Ze} \mid \mathit{Su}\ \mathit{Nat}$$

$$\begin{aligned}&\mathbf{data}\ \mathit{Vec} :: \star \to \mathit{Nat} \to \star\ \mathbf{where}\\ &\quad \mathit{Nil}\ \ \ :: \mathit{Vec}\ a\ \mathit{Ze}\\ &\quad \mathit{Cons} :: a \to \mathit{Vec}\ a\ n \to \mathit{Vec}\ a\ (\mathit{Su}\ n)\end{aligned}$$

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 $\mathit{Vec}\ \mathit{Int}\ \mathit{Int}$ 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:

$$
\begin{aligned}
&\mathbf{data}\ \mathit{Bool} &&= \mathit{True} \mid \mathit{False} \\
&\mathbf{data}\ \mathit{Tree}\ a &&= \mathit{Leaf} \mid \mathit{Bin}\ a\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a) \\
&\mathbf{data}\ \mathit{Rose}\ a &&= \mathit{Rose}\ a\ [\mathit{Rose}\ a] \\
&\mathbf{data}\ \mathit{Perfect}\ a &&= \mathit{Split}\ (\mathit{Perfect}\ (a, a)) \mid \mathit{Element}\ a
\end{aligned}
$$

Datatype promotion II

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

$$
\begin{aligned}
&\mathbf{data}\ \mathit{Bool} &&= \mathit{True} \mid \mathit{False} \\
&\mathbf{data}\ \mathit{Tree}\ a &&= \mathit{Leaf} \mid \mathit{Bin}\ a\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a) \\
&\mathbf{data}\ \mathit{Rose}\ a &&= \mathit{Rose}\ a\ [\mathit{Rose}\ a] \\
&\mathbf{data}\ \mathit{Perfect}\ a &&= \mathit{Split}\ (\mathit{Perfect}\ (a, a)) \mid \mathit{Element}\ a
\end{aligned}
$$

But the following are not promoted:

$$
\begin{aligned}
&\mathbf{data}\ \mathit{Fix}\ f &&= \mathit{In}\ (f\ (\mathit{Fix}\ f)) \\
&\mathbf{data}\ \mathit{Dynamic} &&= \forall t. \mathit{Typeable}\ t \Rightarrow \mathit{Dyn}\ t \\
&\mathbf{data}\ \mathit{Vec} :: \star \to \mathit{Nat} \to \star\ \mathbf{where} \\
&\quad \mathit{Nil}\ \ :: \mathit{Vec}\ a\ \mathit{Ze} \\
&\quad \mathit{Cons} :: a \to \mathit{Vec}\ a\ n \to \mathit{Vec}\ a\ (\mathit{Su}\ n)
\end{aligned}
$$

Datatype promotion III

Type families can also be indexed over promoted types:

$$\begin{array}{l} \textbf{type family}\ \mathit{Add}\ (m :: \mathit{Nat})\ (n :: \mathit{Nat}) :: \mathit{Nat} \\ \textbf{type instance}\ \mathit{Add}\ \mathit{Ze}\ \qquad\quad n = n \\ \textbf{type instance}\ \mathit{Add}\ (\mathit{Su}\ m)\ n = \mathit{Su}\ (\mathit{Add}\ m\ n) \end{array}$$

Datatype promotion III

Type families can also be indexed over promoted types:

$$\textbf{type family } \mathit{Add}\ (m :: \mathit{Nat})\ (n :: \mathit{Nat}) :: \mathit{Nat}$$

$$\begin{aligned}
&\textbf{type instance } \mathit{Add}\ \mathit{Ze} && n = n \\
&\textbf{type instance } \mathit{Add}\ (\mathit{Su}\ m) && n = \mathit{Su}\ (\mathit{Add}\ m\ n)
\end{aligned}$$

$$\begin{aligned}
&\mathit{append} :: \mathit{Vec}\ a\ m \to \mathit{Vec}\ a\ n \to \mathit{Vec}\ a\ (\mathit{Add}\ m\ n) \\
&\mathit{append}\ \mathit{Nil} \qquad\qquad\quad v = v \\
&\mathit{append}\ (\mathit{Cons}\ h\ t)\ v = \mathit{Cons}\ h\ (\mathit{append}\ t\ v)
\end{aligned}$$

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:

$$
\begin{array}{l}
\textbf{data}\ \mathit{HList} :: [\star] \rightarrow \star\ \textbf{where} \\
\quad \mathit{HNil} \quad :: \mathit{HList}\ [] \\
\quad \mathit{HCons} :: a \rightarrow \mathit{HList}\ t \rightarrow \mathit{HList}\ (a : t)
\end{array}
$$

As an example, here is a heterogeneous collection:

$$
\begin{array}{l}
\mathit{hetList} :: \mathit{HList}\ [\mathit{Int}, \mathit{Bool}] \\
\mathit{hetList} = \mathit{HCons}\ 3\ (\mathit{HCons}\ \mathit{False}\ \mathit{HNil})
\end{array}
$$

Tuples are also promoted, e.g. $(\star, \star \rightarrow \star, \mathit{Constraint})$.

Kind-polymorphic type equality

Kind polymorphism reduces code duplication:

$$\begin{array}{l}\mathbf{data}\ Eq_T\ a\ b\ \mathbf{where}\\ \quad \mathit{Refl} :: Eq_T\ a\ a\end{array}$$

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: $Eq_T\ a\ Int$, $Eq_T\ f\ Maybe$, $Eq_T\ t\ Either$.

Kind-polymorphic *Typeable* I

Now we can define a single kind-polymorphic *Typeable* class:

$$\textbf{data}\ \mathit{Proxy}\ (t :: k) = \mathit{Proxy}$$

$$\begin{array}{l}\textbf{class}\ \mathit{Typeable}\ (t :: k)\ \textbf{where}\\ \quad \mathit{typeRep} :: \mathit{Proxy}\ t \to \mathit{TypeRep}\end{array}$$

Note that *Proxy* is kind polymorphic!

Kind-polymorphic *Typeable* II

We can give *Typeable* instances for types of various kinds:

$$\begin{array}{llll}\textbf{instance} & \mathit{Typeable} & \mathit{Char} & \textbf{where}\ldots \\ \textbf{instance} & \mathit{Typeable} & [\,] & \textbf{where}\ldots \\ \textbf{instance} & \mathit{Typeable} & \mathit{Either} & \textbf{where}\ldots\end{array}$$

Kind-polymorphic *Typeable* III

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

$$
\begin{aligned}
&typeOf :: \forall a. Typeable\ a \Rightarrow a \to TypeRep \\
&typeOf\ x = typeRep\ (getType\ x)\ \mathbf{where} \\
&\quad getType :: a \to Proxy\ a \\
&\quad getType\ \_ = Proxy
\end{aligned}
$$

Kind-polymorphic *Typeable* III

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

$$
\begin{array}{l}
typeOf :: \forall a. Typeable\ a \Rightarrow a \rightarrow TypeRep \\
typeOf\ x = typeRep\ (getType\ x)\ \mathbf{where} \\
\quad getType :: a \rightarrow Proxy\ a \\
\quad getType\ \_ = Proxy
\end{array}
$$

$$
\begin{array}{l}
typeOf_1 :: \forall f\ (a :: \star). Typeable\ f \Rightarrow f\ a \rightarrow TypeRep \\
typeOf_1\ x = typeRep\ (getType_1\ x)\ \mathbf{where} \\
\quad getType_1 :: f\ a \rightarrow Proxy\ f \\
\quad getType_1\ \_ = Proxy
\end{array}
$$

2. Type-level literals

Type-level literals

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

$$0, 1, 2, \ldots :: Nat$$

Note the colours!

These type-level naturals come with associated operations:

$$
\begin{aligned}
(\leqslant) &:: Nat \rightarrow Nat \rightarrow Constraint \\
(+) &:: Nat \rightarrow Nat \rightarrow Nat \\
(*) &:: Nat \rightarrow Nat \rightarrow Nat \\
(\hat{}) &:: Nat \rightarrow Nat \rightarrow Nat
\end{aligned}
$$

Value-level reflection

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

$$\mathbf{newtype}\ \mathit{Sing} :: a \to \star$$

Value-level reflection

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

$$\mathbf{newtype}\ \mathit{Sing} :: a \to \star$$

From types to values:

$$\mathit{fromSing} :: \mathit{Sing}\ a \to \mathit{SingRep}\ a$$

$$\begin{array}{l}\mathbf{type\ family}\ \mathit{SingRep}\ a \\ \mathbf{type\ instance}\ \mathit{SingRep}\ (a :: \mathit{Nat}) \qquad = \mathit{Integer} \\ \mathbf{type\ instance}\ \mathit{SingRep}\ (a :: \mathit{Symbol}) = \mathit{String}\end{array}$$

Note that we can have type-level literals other than naturals, and *SingRep* is a kind-indexed family!

Revisiting vectors

Revisiting vectors, now with type-level naturals:

$$
\begin{aligned}
&\mathbf{data}\ \mathit{Vec} :: \mathit{Nat} \to \star \to \star\ \mathbf{where} \\
&\quad \mathit{Nil}\ \ \ :: \mathit{Vec}\ 0\ a \\
&\quad \mathit{Cons} :: a \to \mathit{Vec}\ n\ a \to \mathit{Vec}\ (n+1)\ a
\end{aligned}
$$

Vector concatenation uses type-level natural number addition:

$$
\begin{aligned}
&\mathit{append} :: \mathit{Vec}\ m\ a \to \mathit{Vec}\ n\ a \to \mathit{Vec}\ (m+n)\ a \\
&\mathit{append}\ \mathit{Nil}\ \qquad\qquad ys = ys \\
&\mathit{append}\ (\mathit{Cons}\ x\ xs)\ ys = \mathit{Cons}\ x\ (\mathit{append}\ xs\ ys)
\end{aligned}
$$

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 $Nil :: \forall a. Vec\ 0\ a$
- Could not deduce $((m + n) \sim ((n' + n) + 1))$ from the context $(m \sim (n' + 1))$ bound by a pattern with constructor $Cons :: \forall a\ (n :: Nat). a \rightarrow Vec\ n\ a \rightarrow Vec\ (n + 1)\ a$

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?

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 I

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

$$
\begin{aligned}
&p, q :: \color{red}{Int} \\
&p = 1 \\
&q = \texttt{'1'} \\
&main = print\ p
\end{aligned}
$$

Compiles with warning: “couldn’t match expected type $\color{red}{Int}$ with actual type $\color{red}{Char}$ in an equation for q: $q = \texttt{'1'}$”.

Runs and returns 1.

Example II

$$
\begin{aligned}
&p, q :: Int \\
&p = 1 \\
&q = \texttt{'1'} \\
&main = print\ q
\end{aligned}
$$

Fails at runtime with: "couldn't match expected type Int with actual type $Char$ in an equation for q: $q = \texttt{'1'}$".

Example III

$$
\begin{aligned}
&t_1 :: Int\\
&t_1 = \texttt{'1'}\\
&t_2 :: a \to String\\
&t_2 = show\\
&\textbf{data}\ T\ a\ \textbf{where}\\
&\quad T_1 :: Int \to T\ Int\\
&\quad T_2 :: a \quad\ \to T\ a\\
&t_3 :: T\ a\\
&t_3 = T_1\ 0\\
&\\
&main = print\ 1
\end{aligned}
$$

Runs fine!

How it works

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

$$\mathbf{data}\ T\ a = T_1\ (a \sim Int)\ Int \mid T_2\ a$$

$$unT :: T\ a \to a$$
$$unT\ (T_1\ c\ n) = n\ \triangleright\ (sym\ c)$$
$$unT\ (T_2\ x) \quad = x$$

$$\triangleright :: b \to (b \sim a) \to a$$

Evidence, or values of type $(\sim)$, 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:

$$f :: \forall a . a \to a \to a$$
$$f\ x\ y = x \wedge y$$
$$main = print\ (f\ \mathit{True}\ \mathit{False})$$

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

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?