Universiteit Utrecht

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

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

Introduction

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'`

Colors

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

Colors

In this talk I will use:

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

What are kinds?

Just like types classify values...

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

What are kinds?

Just like types classify values...

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

... kinds classify types:

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

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 *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:

$$\begin{aligned}
&\textbf{data}\ Ze \\
&\textbf{data}\ Su\ n
\end{aligned}$$

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

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!

Universiteit Utrecht
[Faculty of Science
Information and Computing Sciences]

Why do we need a better kind system? II

Lack of kind polymorphism leads to code duplication:

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

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

Datatype promotion I

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

$$
\begin{array}{l}
\mathbf{data}\ Nat = Ze \mid Su\ Nat \\
\mathbf{data}\ Vec :: \star \to Nat \to \star\ \mathbf{where} \\
\quad Nil \quad :: Vec\ a\ Ze \\
\quad Cons :: a \to Vec\ a\ n \to Vec\ a\ (Su\ n)
\end{array}
$$

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:

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

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:

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

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:

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

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

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

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:

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

As an example, here is a heterogeneous collection:

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

Promoted lists and tuples

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

$$\begin{aligned}
&\mathbf{data}\ \mathit{HList} :: [\star] \to \star\ \mathbf{where} \\
&\quad \mathit{HNil} \quad :: \mathit{HList}\ [] \\
&\quad \mathit{HCons} :: a \to \mathit{HList}\ t \to \mathit{HList}\ (a : t)
\end{aligned}$$

As an example, here is a heterogeneous collection:

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

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

Kind-polymorphic type equality

Kind polymorphism reduces code duplication:

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

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

Kind-polymorphic *Typeable* I

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

$$\textbf{data}\ \mathit{Proxy}\ t = \mathit{Proxy}$$

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

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:

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

Kind-polymorphic *Typeable* III

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

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

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

Kind-polymorphic fixed-point operator

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

$$\textbf{data}\ \mathit{Mu}\ f\ \ a = \mathit{Roll}\ (f\ \ (\mathit{Mu}\ f)\ \ a)$$

Kind-polymorphic fixed-point operator

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

$$\mathbf{data}\ \mathit{Mu}\ f\ a = \mathit{Roll}\ (f\ (\mathit{Mu}\ f)\ a)$$

Here Mu is instantiated to kind $((\star \to \star) \to \star \to \star) \to \star \to \star$:

$$\mathbf{data}\ \mathit{ListF}\ f\ a = \mathit{NilF} \mid \mathit{ConsF}\ a\ (f\ a)$$
$$\mathbf{type}\ \mathit{List}\ a = \mathit{Mu}\ \mathit{ListF}\ a$$

Kind-polymorphic fixed-point operator

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

$$\mathbf{data}\ \mathit{Mu}\ f\ a = \mathit{Roll}\ (f\ (\mathit{Mu}\ f)\ a)$$

Here Mu is instantiated to kind $((\star \to \star) \to \star \to \star) \to \star \to \star$:

$$\mathbf{data}\ \mathit{ListF}\ f\ a = \mathit{NilF} \mid \mathit{ConsF}\ a\ (f\ a)$$
$$\mathbf{type}\ \mathit{List}\ a = \mathit{Mu}\ \mathit{ListF}\ a$$

And here to kind $((\mathit{Nat} \to \star) \to \mathit{Nat} \to \star) \to \mathit{Nat} \to \star$:

$$\begin{array}{l}\mathbf{data}\ \mathit{VecF}\ (a :: \star)\ (f :: \mathit{Nat} \to \star)\ (n :: \mathit{Nat})\ \mathbf{where} \\ \quad \mathit{VFNil} \quad :: \mathit{VecF}\ a\ f\ \mathit{Ze} \\ \quad \mathit{VFCons} :: a \to f\ n \to \mathit{VecF}\ a\ f\ (\mathit{Su}\ n) \\ \mathbf{type}\ \mathit{Vec}\ a\ n = \mathit{Mu}\ (\mathit{VecF}\ a)\ n\end{array}$$

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.

$$
\begin{aligned}
\textbf{data}\ \mathit{Universe}\ x = {} & \underline{U} \\
\mid {} & \underline{K}\ x \\
\mid {} & \mathit{Universe}\ x\ \underline{+}\ \mathit{Universe}\ x \\
\mid {} & \mathit{Universe}\ x\ \underline{\times}\ \mathit{Universe}\ x
\end{aligned}
$$

Better kinded generics: representation

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

$$
\begin{aligned}
\textbf{data}\ \mathit{Universe}\ x = {} & \underline{U} \\
\mid {} & \underline{K}\ x \\
\mid {} & \mathit{Universe}\ x\ \underline{+}\ \mathit{Universe}\ x \\
\mid {} & \mathit{Universe}\ x\ \underline{\times}\ \mathit{Universe}\ x
\end{aligned}
$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{data}\ \mathit{Interprt} :: \mathit{Universe}\ \star \to \star\ \textbf{where}} \\
\quad U & :: & \mathit{Interprt}\ \underline{U} \\
\quad K & :: t & \to \mathit{Interprt}\ (\underline{K}\ t) \\
\quad L & :: \mathit{Interprt}\ a & \to \mathit{Interprt}\ (a\ \underline{+}\ b) \\
\quad R & :: \mathit{Interprt}\ b & \to \mathit{Interprt}\ (a\ \underline{+}\ b) \\
\quad (\times) & :: \mathit{Interprt}\ a \to \mathit{Interprt}\ b & \to \mathit{Interprt}\ (a\ \underline{\times}\ b)
\end{array}
$$

Better kinded generics: conversion

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

$$
\begin{aligned}
&\textbf{class}\ \mathit{Representable}\ a\ \textbf{where} \\
&\quad \textbf{type}\ \mathit{Rep}\ a :: \mathit{Universe}\ \star \\
&\quad \mathit{from} :: a \to \mathit{Interprt}\ (\mathit{Rep}\ a) \\
&\quad \mathit{to}\quad\ :: \mathit{Interprt}\ (\mathit{Rep}\ a) \to a
\end{aligned}
$$

Better kinded generics: conversion

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

$$
\begin{aligned}
&\textbf{class}\ \mathit{Representable}\ a\ \textbf{where}\\
&\quad \textbf{type}\ \mathit{Rep}\ a :: \mathit{Universe}\ \star\\
&\quad \mathit{from} :: a \rightarrow \mathit{Interprt}\ (\mathit{Rep}\ a)\\
&\quad \mathit{to} \quad :: \mathit{Interprt}\ (\mathit{Rep}\ a) \rightarrow a
\end{aligned}
$$

User types are (automatically) representable:

$$
\begin{aligned}
&\textbf{instance}\ \mathit{Representable}\ [a] &&\textbf{where}\ldots\\
&\textbf{instance}\ \mathit{Representable}\ (\mathit{Maybe}\ a) &&\textbf{where}\ldots\\
&\textbf{instance}\ \mathit{Representable}\ (\mathit{Either}\ a\ b) &&\textbf{where}\ldots
\end{aligned}
$$

Better kinded generics: functions

User-visible class, exported:

$$
\begin{array}{l}
\textbf{class}\ \mathit{Show}\ (a :: \star)\ \textbf{where} \\
\quad \mathit{show} :: a \rightarrow \mathit{String} \\
\quad \textbf{default}\ \mathit{show}\ :: (\mathit{Representable}\ a, \mathit{GShow}\ (\mathit{Rep}\ a)) \\
\qquad\qquad\qquad\quad \Rightarrow a \rightarrow \mathit{String} \\
\quad \mathit{show} = \mathit{gshow} \circ \mathit{from}
\end{array}
$$

Better kinded generics: functions

User-visible class, exported:

$$
\begin{aligned}
&\mathbf{class}\ \mathit{Show}\ (a :: \star)\ \mathbf{where} \\
&\quad \mathit{show} :: a \to \mathit{String} \\
&\quad \mathbf{default}\ \mathit{show}\ :: (\mathit{Representable}\ a, \mathit{GShow}\ (\mathit{Rep}\ a)) \\
&\qquad\qquad\qquad \Rightarrow a \to \mathit{String} \\
&\quad \mathit{show} = \mathit{gshow} \circ \mathit{from}
\end{aligned}
$$

Defined by the generic programmer, not exported:

$$
\begin{aligned}
&\mathbf{class}\ \mathit{GShow}\ (a :: \mathit{Universe}\ \star)\ \mathbf{where} \\
&\quad \mathit{gshow} :: \mathit{Interprt}\ a \to \mathit{String} \\
&\mathbf{instance}\ \mathit{GShow}\ \underline{U}\ \mathbf{where} \\
&\quad \mathit{gshow}\ U = \texttt{""} \\
&\ldots
\end{aligned}
$$

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

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