Type Families in Haskell

Manuel M. T. Chakravarty
University of New South Wales

Joint work with
Gabriele Keller
Simon Peyton Jones
Simon Marlow

. . . and more recently
Tom Schrijvers
Martin Sulzmann

Motivation

Type classes

- Most innovative feature of Haskell
- Proved useful beyond simple overloading of equality, ordering, and arithmetic functions

Motivation

Type classes

- Most innovative feature of Haskell
- Proved useful beyond simple overloading of equality, ordering, and arithmetic functions

Multi-parameter type classes

- Haskell 98 permits only a single parameter to a type class
- Multiple parameters can require a lot of type annotations
- Functional dependencies were proposed as a solution:
  - Led to a lot of interesting type level programming
  - But their syntax is relational, not functional
  - And they have limits

PLS

Motivation

Type classes

- Most innovative feature of Haskell
- Proved useful beyond simple overloading of equality, ordering, and arithmetic functions

Multi-parameter type classes

- Haskell 98 permits only a single parameter to a type class
- Multiple parameters can require a lot of type annotations
- Functional dependencies were proposed as a solution:
  - Led to a lot of interesting type level programming
  - But their syntax is relational, not functional
  - And they have limits

C++ success story

- `typedefs` in classes $\Rightarrow$ traits classes in the STL

Type Classes in a Nutshell

Ad-hoc polymorphism (overloading)

$$\textbf{class}\ Eq\ a\ \textbf{where}$$
$$\quad (==)\ ::\ a \to a \to Bool$$

Ad-hoc polymorphism (overloading)

class *Eq a* **where**
  $(==) \; :: \; a \; \to \; a \; \to \; Bool$
instance *Eq Int* **where**
  $(==) \; = \; primEqInt$

Type Classes in a Nutshell

Ad-hoc polymorphism (overloading)

$$
\begin{array}{l}
\textbf{class } \mathit{Eq}\ a \textbf{ where} \\
\quad (==) :: a \to a \to \mathit{Bool} \\
\textbf{instance } \mathit{Eq}\ \mathit{Int} \textbf{ where} \\
\quad (==) = \mathit{primEqInt} \\
\textbf{instance } (\mathit{Eq}\ a, \mathit{Eq}\ b) \Rightarrow \mathit{Eq}\ (a, b) \textbf{ where} \\
\quad (x_1, y_1) == (x_2, y_2) = (x_1 == x_2)\ \&\&\ (y_1 == y_2)
\end{array}
$$

PLS

Type Classes in a Nutshell

Ad-hoc polymorphism (overloading)

$$
\begin{array}{l}
\textbf{class } Eq\ a \textbf{ where} \\
\quad (==) :: a \rightarrow a \rightarrow Bool \\
\textbf{instance } Eq\ Int \textbf{ where} \\
\quad (==) = primEqInt \\
\textbf{instance } (Eq\ a, Eq\ b) \Rightarrow Eq\ (a, b) \textbf{ where} \\
\quad (x_1, y_1) == (x_2, y_2) = (x_1 == x_2)\ \&\&\ (y_1 == y_2)
\end{array}
$$

Full type of equality is. . .

$$(==) :: Eq\ a \Rightarrow a \rightarrow a \rightarrow Bool \qquad \text{-- qualified type}$$

Usage: $(2, (3, 4)) == (2, (3, 4))$

Why Type Families?

A motivating programming problem

- Family of containers with different representation types (e.g., lists, trees, arrays, bit sets)
- Representation type determines the element type plus additional constraints

Why Type Families?

A motivating programming problem

- Family of containers with different representation types (e.g., lists, trees, arrays, bit sets)
- Representation type determines the element type plus additional constraints

Type of the insertion function

$$insert :: Collects\ c \Rightarrow Elem\ c \rightarrow c \rightarrow c$$

where

- $Collects\ c$ asserts that c represents a collection
- $Elem\ c$ maps c to its element type

For example,

$$\begin{array}{lll} Elem\ [e] & = e & \text{for } Collects\ [e] \\ Elem\ BitSet & = Char & \text{for } Collects\ BitSet \end{array}$$

With associated type synonym families

$$
\begin{array}{l}
\textbf{class}\ \mathit{Collects}\ c\ \textbf{where} \\
\quad \mathit{empty} :: c \\
\quad \mathit{insert} :: \mathit{Elem}\ c \rightarrow c \rightarrow c \\
\quad \mathit{toList} :: c \rightarrow [\mathit{Elem}\ c] \\
\textbf{instance}\ \mathit{Eq}\ e \Rightarrow \mathit{Collects}\ [e]\ \textbf{where} \\
\quad \ldots \\
\textbf{instance}\ \mathit{Collects}\ \mathit{BitSet}\ \textbf{where} \\
\quad \ldots \\
\textbf{instance}\ (\mathit{Collects}\ c, \mathit{Hashable}\ (\mathit{Elem}\ c)) \Rightarrow \\
\qquad \mathit{Collects}\ (\mathit{Array}\ \mathit{Int}\ c)\ \textbf{where} \\
\quad \ldots
\end{array}
$$

With associated type synonym families

$$
\begin{array}{l}
\mathbf{class}\ \mathit{Collects}\ c\ \mathbf{where} \\
\quad \mathbf{type}\ \ \mathit{Elem}\ c \qquad\qquad\qquad\qquad \text{-- definition varies with } c \\
\quad \mathit{empty} :: c \\
\quad \mathit{insert} :: \mathit{Elem}\ c \to c \to c \\
\quad \mathit{toList} :: c \to [\mathit{Elem}\ c] \\
\\
\mathbf{instance}\ \mathit{Eq}\ e \Rightarrow \mathit{Collects}\ [e]\ \mathbf{where} \\
\quad \mathbf{type}\ \mathit{Elem}\ [e] = e \\
\quad \ldots \\
\mathbf{instance}\ \mathit{Collects}\ \mathit{BitSet}\ \mathbf{where} \\
\quad \mathbf{type}\ \mathit{Elem}\ \mathit{BitSet} = \mathit{Char} \\
\quad \ldots \\
\mathbf{instance}\ (\mathit{Collects}\ c, \mathit{Hashable}\ (\mathit{Elem}\ c)) \Rightarrow \\
\qquad \mathit{Collects}\ (\mathit{Array}\ \mathit{Int}\ c)\ \mathbf{where} \\
\quad \mathbf{type}\ \mathit{Elem}\ (\mathit{Array}\ \mathit{Int}\ c) = \mathit{Elem}\ c \\
\quad \ldots
\end{array}
$$

$$
\begin{array}{l}
\textbf{class } \mathit{Collects}\ c \textbf{ where} \\
\quad \textbf{type } \mathit{Elem}\ c \\
\quad \mathit{empty} :: c \\
\quad \mathit{insert} :: \mathit{Elem}\ c \to c \to c \\
\quad \mathit{toList} :: c \to [\mathit{Elem}\ c]
\end{array}
$$

$\mathit{foldr} :: (a \to b \to b) \to b \to [a] \to b$ -- standard function

Make a collection from a list of elements

$$
\begin{array}{l}
\mathit{fromList} :: ??? \\
\mathit{fromList}\ l = \mathit{foldr}\ \mathit{insert}\ \mathit{empty}\ l
\end{array}
$$

$$
\begin{array}{l}
\textbf{class } \mathit{Collects}\ c \textbf{ where} \\
\quad \textbf{type } \mathit{Elem}\ c \\
\quad \mathit{empty} :: c \\
\quad \mathit{insert} :: \mathit{Elem}\ c \to c \to c \\
\quad \mathit{toList} :: c \to [\mathit{Elem}\ c]
\end{array}
$$

$\mathit{foldr} :: (a \to b \to b) \to b \to [a] \to b$ -- standard function

Make a collection from a list of elements

$$
\begin{array}{l}
\mathit{fromList} :: \mathit{Collects}\ c \Rightarrow [\mathit{Elem}\ c] \to c \\
\mathit{fromList}\ l = \mathit{foldr}\ \mathit{insert}\ \mathit{empty}\ l
\end{array}
$$

Merge elements of one collection into another

$$\begin{aligned} & merge :: (Collects\ c1,\ Collects\ c2,\ ????) \\ & \qquad \Rightarrow c1 \to c2 \to c2 \\ & merge\ c1\ c2 = foldr\ insert\ c2\ (toList\ c1) \end{aligned}$$

Make a collection from a list of elements

$$\begin{aligned} & fromList :: Collects\ c \Rightarrow [Elem\ c] \to c \\ & fromList\ l = foldr\ insert\ empty\ l \end{aligned}$$

Merge elements of one collection into another

$$
\begin{aligned}
&\mathit{merge} :: (\mathit{Collects}\ c1,\ \mathit{Collects}\ c2,\ \mathit{Elem}\ c1 \sim \mathit{Elem}\ c2) \\
&\qquad \Rightarrow c1 \to c2 \to c2 \\
&\mathit{merge}\ c1\ c2 = \mathit{foldr}\ \mathit{insert}\ c2\ (\mathit{toList}\ c1)
\end{aligned}
$$

- We need equality constraints

Make a collection from a list of elements

$$
\begin{aligned}
&\mathit{fromList} :: \mathit{Collects}\ c \Rightarrow [\mathit{Elem}\ c] \to c \\
&\mathit{fromList}\ l = \mathit{foldr}\ \mathit{insert}\ \mathit{empty}\ l
\end{aligned}
$$

Type Indexed Families of Types

Haskell 98 type classes define families of values

Overloaded functions are typed-indexed families of values:

$$
\begin{gathered}
(+) \;::\; \mathit{Num}\, a \;\Rightarrow\; a \;\rightarrow\; a \;\rightarrow\; a \\
\approx \\
\left\{
\begin{array}{c}
\mathit{addInt} :: \mathit{Int} \rightarrow \mathit{Int} \rightarrow \mathit{Int} \\
\mathit{addFloat} :: \mathit{Float} \rightarrow \mathit{Float} \rightarrow \mathit{Float} \\
\vdots
\end{array}
\right\}
\end{gathered}
$$

Type Indexed Families of Types

Haskell 98 type classes define families of values

Overloaded functions are typed-indexed families of values:

$$(+)\ ::\ Num\,a\ \Rightarrow\ a\ \rightarrow\ a\ \rightarrow\ a$$
$$\approx$$
$$\left\{\begin{array}{c} addInt :: Int \rightarrow Int \rightarrow Int \\ addFloat :: Float \rightarrow Float \rightarrow Float \\ \vdots \end{array}\right\}$$

We add families of types

Typed-indexed families of types map index types to family members:

$$Elem\ ::\ \star\ \rightarrow\ \star \qquad \approx \qquad \left\{\begin{array}{c} Elem\ [e] = e \\ Elem\ BitSet = Char \\ \vdots \end{array}\right\}$$

Type families need not be associated

- We associated the family *Elem* with the class *Collects*
- Such associations are often convenient, but they are not essential (familiy declarations in classes are just sugar)

Type families need not be associated

- We associated the family $Elem$ with the class $Collects$
- Such associations are often convenient, but they are not essential (familiy declarations in classes are just sugar)

Bounded lists

$$
\begin{array}{ll}
\textbf{data}\ Zero;\ \textbf{data}\ Succ\ a; & \text{-- empty data type representing} \\
& \text{-- Peano numbers as types}
\end{array}
$$

-- adding type numbers

$$
\begin{array}{lcl}
\textbf{type family}\ Add & :: & \star \to \star \to \star \\
\textbf{type instance}\ Add\ Zero\ y & = & y \\
\textbf{type instance}\ Add\ (Succ\ x)\ y & = & Succ\ (Add\ x\ y)
\end{array}
$$

$$
\begin{array}{ll}
\textbf{data}\ BList\ n\ a\ \textbf{where} & \text{-- bounded lists as GADT} \\
\quad BNil\ \ :: BList\ Zero\ a & \\
\quad BCons :: a \to BList\ n\ a \to BList\ (Succ\ n)\ a &
\end{array}
$$

Data Type Families

Unboxed arrays

- Boxed array: array of pointers to heap objects
- Unboxed array: array of basic types (as in C)

Data Type Families

Unboxed arrays

- Boxed array: array of pointers to heap objects
- Unboxed array: array of basic types (as in C)

Flattened arrays

Array representation depends on the element type:

$$
\begin{array}{lll}
\textbf{data family}\ \mathit{Array}\ e & & \text{-- family declaration (lifted)} \\
\textbf{data instance}\ \mathit{Array}\ \mathit{Int} & = & \mathit{IntArr}\ \mathit{UnboxedIntArr} \\
\textbf{data instance}\ \mathit{Array}\ \mathit{Float} & = & \mathit{IntArr}\ \mathit{UnboxedFloatArr} \\
\textbf{data instance}\ \mathit{Array}\ (a,\ b) & = & \mathit{PairArr}\ (\mathit{Array}\ a)\ (\mathit{Array}\ b)
\end{array}
$$

Data Type Families

Unboxed arrays

- Boxed array: array of pointers to heap objects
- Unboxed array: array of basic types (as in C)

Flattened arrays

Array representation depends on the element type:

$$
\begin{array}{lll}
\textbf{data family}\ \mathit{Array}\ e & & \text{-- family declaration (lifted)} \\
\textbf{data instance}\ \mathit{Array}\ \mathit{Int} & = & \mathit{IntArr}\ \mathit{UnboxedIntArr} \\
\textbf{data instance}\ \mathit{Array}\ \mathit{Float} & = & \mathit{IntArr}\ \mathit{UnboxedFloatArr} \\
\textbf{data instance}\ \mathit{Array}\ (a,\ b) & = & \mathit{PairArr}\ (\mathit{Array}\ a)\ (\mathit{Array}\ b) \\
\textbf{data instance}\ \mathit{Array}\ (\mathit{Array}\ a) & = & \mathit{ArrArr}\ \mathit{Segd}\ (\mathit{Array}\ a) \\
\textbf{type}\ \mathit{Segd} & = & \mathit{Array}\ \mathit{Int}
\end{array}
$$

$$[{:}[{:}1, 2{:}],\ [{::}],\ [{:}3, 4, 5{:}]{:}] \ \Rightarrow\ \mathit{ArrArr}\ [{:}2, 0, 3{:}]\ [{:}1, 2, 3, 4, 5{:}]$$

A fairly wild idea: Class Families

John Hughes' Restricted Data Types

The following general set API type is too general:

class *Set s* **where**

$$
\begin{array}{lll}
empty & :: & s\,a \\
insert & :: & a \rightarrow s\,a \rightarrow s\,a
\end{array}
$$

A fairly wild idea: Class Families

John Hughes' Restricted Data Types

The following general set API type is too general:

$$\textbf{class}\ \mathit{Set}\ s\ \textbf{where}$$
$$\begin{aligned}
&\mathit{empty} &&::\ s\ a\\
&\mathit{insert} &&::\ a \rightarrow s\ a \rightarrow s\ a
\end{aligned}$$

Sets as lists (finite maps) requires Eq (Ord) of elements!

$$\textbf{instance}\ \mathit{Set}\ [\,]\ \textbf{where}$$
$$\begin{aligned}
&\mathit{empty} &&= [\,]\\
&\mathit{insert}\ x\ s \mid x\ \text{`}\mathit{elem}\text{`}\ s &&= s\\
&\qquad\qquad\ \mid \mathit{otherwise} &&= x : s
\end{aligned}$$

A fairly wild idea: Class Families

John Hughes' Restricted Data Types

The following general set API type is too general:

$$
\begin{array}{ll}
\textbf{class } \mathit{Set}\ s \textbf{ where} & \\
\quad \textbf{class } \mathit{Restrict}\ s\ a & \text{-- associated class indexed by } s \\
\quad \mathit{empty} & ::\ s\ a \\
\quad \mathit{insert} & ::\ \mathit{Restrict}\ s\ a \Rightarrow a \rightarrow s\ a \rightarrow s\ a
\end{array}
$$

Associated class families to the rescue!

Sets as lists (finite maps) requires Eq (Ord) of elements!

$$
\begin{array}{lll}
\textbf{instance } \mathit{Set}\ [\,] \textbf{ where} & & \\
\quad \textbf{class } \mathit{Eq}\ a \Rightarrow \mathit{Restrict}\ [\,]\ a & & \\
\quad \mathit{empty} & = & [\,] \\
\quad \mathit{insert}\ x\ s \mid x\ \text{`}\mathit{elem}\text{`}\ s & = & s \\
\quad \phantom{\mathit{insert}\ x\ s} \mid \mathit{otherwise} & = & x : s \\
\textbf{instance } \mathit{Eq}\ a \Rightarrow \mathit{Restrict}\ [\,]\ a & & \text{-- Tiresome instance}
\end{array}
$$

Implementation Status

Where are we right now?

- Data families
  - Fully implemented in GHC 6.7
- Synonym families
  - Partially implemented; working at it
  - We think we know how to perform type inference with type families and GADTs
- Class families
  - Just an idea at this stage (should be easy to implement)

Further information

- User manual
  `http://haskell.org/haskellwiki/GHC/Indexed_types`
- Implementation notes
  `http://hackage.haskell.org/trac/ghc/wiki/TypeFunctions`