Type Families: Program Your Types!

Manuel M. T. Chakravarty¹
Simon Peyton Jones²
Tom Schrijvers³
Martin Sulzmann⁴
Gabriele Keller¹
Simon Marlow²

¹University of New South Wales, Sydney
²Microsoft Research Ltd, Cambridge
³Katholieke Universiteit Leuven
⁴IT University of Copenhagen

(Work partially funded by the Australian Research Council.)
In the beginning, there was this itch...

While working on nested data parallelism for Haskell

- High-performance arrays (same ballpark as C/C++)
- Parametric interface (should support arrays of trees etc.)
Boxed arrays

\[ \text{data } \text{Array } a \quad \text{-- parametric arrays} \]

- generic representation
- storage intensive & cache unfriendly
Boxed arrays

\[ \text{data} \quad \text{Array} \ a \quad -- \ \text{parametric arrays} \]

- generic representation
- storage intensive & cache unfriendly

Example

\[ \text{Array} \ (\text{Int}, \text{Bool}) \]

Boxed representation
Unboxed arrays

-data family $\text{Array } a$ -- type-indexed arrays
-data instance $\text{Array } \text{Int} = \text{IntArr UnboxedIntArr}$
-data instance $\text{Array } \text{Bool} = \text{BoolArr UnboxedBitVector}$
-data instance $\text{Array } (a, b) = \text{PairArr } (\text{Array } a) (\text{Array } b)$

Example

$\text{Array } (\text{Int}, \text{Bool})$

Boxed representation
Unboxed arrays

\[\text{data family } \text{Array } a\] -- type-indexed arrays
\[\text{data instance } \text{Array } \text{Int} = \text{IntArr } \text{UnboxedIntArr}\]
\[\text{data instance } \text{Array } \text{Bool} = \text{BoolArr } \text{UnboxedBitVector}\]
\[\text{data instance } \text{Array } (a, b) = \text{PairArr } (\text{Array } a) (\text{Array } b)\]

Example

\[\text{Array } (\text{Int}, \text{Bool})\]

Unboxed representation
Unboxed arrays

\[
\begin{align*}
\text{data family} & \quad \text{Array } a & \quad \text{-- type-indexed arrays} \\
\text{data instance} & \quad \text{Array } \text{Int} & = \quad \text{IntArr UnboxedIntArr} \\
\text{data instance} & \quad \text{Array } \text{Bool} & = \quad \text{BoolArr UnboxedBitVector} \\
\text{data instance} & \quad \text{Array } (a, b) & = \quad \text{PairArr} (\text{Array } a) (\text{Array } b) \\
\text{data instance} & \quad \text{Array } (\text{Array } a) & = \quad \text{ArrArr} (\text{Array } \text{Int}) (\text{Array } a)
\end{align*}
\]

Example

\[
\text{Array } (\text{Int}, \text{Bool})
\]

\[
\begin{array}{c}
\begin{array}{cccccccccc}
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\end{array} \\
\begin{array}{cccccccccc}
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\end{array} \\
\begin{array}{cccccccccc}
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\end{array} \\
\begin{array}{cccccccccc}
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\end{array} \\
\begin{array}{cccccccccc}
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\end{array} \\
\end{array}
\]

Unboxed representation

\[
\begin{align*}
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\end{align*}
\]

\[
\text{\textbullet [ } [ :1, 2:], [::], [ :3, 4, 5]: ] } \Rightarrow \text{ArrArr } [ :2, 0, 3:] [ :1, 2, 3, 4, 5:] \]
Type-indexed data families: example use case

- Generic data types
- Optimises data representation guided by the type structure
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
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

\[
\text{insert} :: \text{Collects } c \Rightarrow \text{Elem } c \to c \to c
\]

where

- \(\text{Collects } c\) asserts that \(c\) represents a collection
- \(\text{Elem } c\) maps \(c\) to its element type

For example,

\[
\text{Elem } [e] = e \quad \text{for } \text{Collects } [e]
\]
\[
\text{Elem BitSet} = \text{Char} \quad \text{for } \text{Collects BitSet}
\]
With associated type synonym families

```haskell
class Collects c where

    empty :: c
    insert :: Elem c -> c -> c
    toList :: c -> [Elem c]

instance Eq e => Collects [e] where

instance Collects BitSet where

instance (Collects c, Hashable (Elem c)) => Collects (Array Int c) where

...
With associated type synonym families

```haskell
class Collects c where  -- definition varies with c
  type Elem c
  empty :: c
  insert :: Elem c → c → c
  toList :: c → [Elem c]

instance Eq e ⇒ Collects [e] where
  type Elem [e] = e

instance Collects BitSet where
  type Elem BitSet = Char

instance (Collects c, Hashable (Elem c)) ⇒ Collects (Array Int c) where
  type Elem (Array Int c) = Elem c
  ...
```

Manuel Chakravarty  Type Families: Program Your Types!
class Collects c where

  type Elem c
  empty :: c
  insert :: Elem c \rightarrow c \rightarrow c
  toList :: c \rightarrow [Elem c]

foldr :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b -- standard function

Make a collection from a list of elements

  fromList :: ???
  fromList l = foldr insert empty l
class Collects c where
  type Elem c
  empty :: c
  insert :: Elem c → c → c
  toList :: c → [Elem c]

foldr :: (a → b → b) → b → [a] → b  -- standard function

Make a collection from a list of elements

fromList :: Collects c ⇒ [Elem c] → c
fromList l = foldr insert empty l
Merge elements of one collection into another

\[ \text{merge} :: (\text{Collects } c_1, \text{Collects } c_2, \text{????}) \Rightarrow c_1 \rightarrow c_2 \rightarrow c_2 \]
\[ \text{merge } c_1 c_2 = \text{foldr insert } c_2 \ (\text{toList } c_1) \]

Make a collection from a list of elements

\[ \text{fromList} :: \text{Collects } c \Rightarrow [\text{Elem } c] \rightarrow c \]
\[ \text{fromList } l = \text{foldr insert empty } l \]
Merge elements of one collection into another

\[ \text{merge} \colon (\text{Collects } c_1, \text{Collects } c_2, \text{Elem } c_1 \sim \text{Elem } c_2) \Rightarrow c_1 \rightarrow c_2 \rightarrow c_2 \]
\[ \text{merge } c_1 \, c_2 = \text{foldr insert } c_2 (\text{toList } c_1) \]

- We need equality constraints

Make a collection from a list of elements

\[ \text{fromList} \colon \text{Collects } c \Rightarrow [\text{Elem } c] \rightarrow c \]
\[ \text{fromList } l = \text{foldr insert empty } l \]
Type-indexed type families: use case #1

- Generic data types with associated types
- Much like traits classes in C++
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 (family 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 (family declarations in classes are just sugar)

Bounded lists

```haskell
data Zero; data Succ a;

-- empty data type representing
-- Peano numbers as types

-- adding type numbers

type family Add :: ⋆ → ⋆ → ⋆
type instance Add Zero y = y
type instance Add (Succ x) y = Succ (Add x y)

data BList n a where

BNil :: BList Zero a
BCons :: a → BList n a → BList (Succ n) a
```

Manuel Chakravarty
Type Families: Program Your Types!
Type families need not be associated
- We associated the family \textit{Elem} with the class \textit{Collects}
- Such associations are often convenient, but they are \textbf{not essential} (family declarations in classes are just sugar)

Bounded lists

\texttt{data Zero; data Succ a;}  -- empty data type representing
\texttt{-- Peano numbers as types}

\texttt{-- adding type numbers}
\texttt{type family Add :: ⋆ → ⋆ → ⋆}
\texttt{type instance Add Zero y = y}
\texttt{type instance Add (Succ x) y = Succ (Add x y)}

\texttt{data BList n a where}  -- bounded lists as GADT
\texttt{BNil :: BList Zero a}
\texttt{BCons :: a → BList n a → BList (Succ n) a}
\texttt{appendBList :: BList n a → BList m a → BList (Add n m) a}
Type-indexed type families: use case #2

- Type-level computations
- Express complex properties as types:
  - Bounded lists
  - Type-preserving compiler (Louis-Julien Guillemette & Stefan Monnier)
- Embedded domain-specific type systems:
  - Andrew Appleyard’s Salsa (.NET bridge)
  - Embedded C# overload resolution in Haskell
From a type-theoretic point of view

- They are equivalent in expressive power
- We have translations in both directions
Type Families and Functional Dependencies

From a type-theoretic point of view
- They are equivalent in expressive power
- We have translations in both directions

From a functional programmer’s point of view
- We are functional, not logic programmers
- FDs require a relational programming style
- TFs enable functional programming on the type-level
- Some contexts (e.g., newtypes) permit TFs, but not FDs
Type Families and Functional Dependencies

From a type-theoretic point of view

- They are equivalent in expressive power
- We have translations in both directions

From a functional programmer’s point of view

- We are functional, not logic programmers
- FDs require a relational programming style
- TFs enable functional programming on the type-level
- Some contexts (e.g., newtypes) permit TFs, but not FDs

From a compiler writer’s point of view

- Type families work fine together with generalised abstract data types (GADTs)
- Type checking with FDs and GADTs is an open problem [well, we could translate the FDs into TFs first...]
In conclusion...

Implementation
- Fully supported in GHC 6.10.1
- Get the release candidate today!

Documentation
http://haskell.org/haskellwiki/GHC/Type_families