

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

data Array a -- parametric arrays

- generic representation
- storage intensive & cache unfriendly

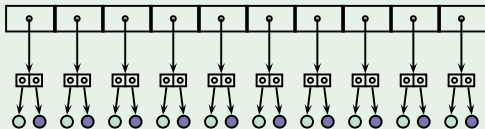
Boxed arrays

data Array *a* -- parametric arrays

- generic representation
- storage intensive & cache unfriendly

Example

Array (Int, Bool)



Boxed representation

Unboxed arrays

data family Array *a*

-- type-indexed arrays

data instance Array Int

= IntArr UnboxedIntArr

data instance Array Bool

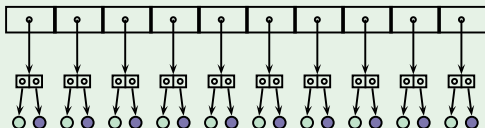
= BoolArr UnboxedBitVector

data instance Array (a, b)

= PairArr (Array a) (Array b)

Example

Array (Int, Bool)



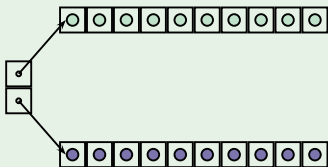
Boxed representation

Unboxed arrays

data family Array *a* -- type-indexed arrays
data instance Array Int = IntArr UnboxedIntArr
data instance Array Bool = BoolArr UnboxedBitVector
data instance Array (a, b) = PairArr (Array a) (Array b)

Example

Array (Int, Bool)



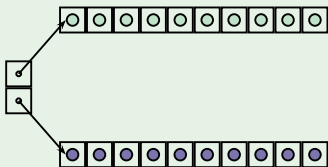
Unboxed representation

Unboxed arrays

data family `Array a` -- type-indexed arrays
data instance `Array Int` = `IntArr UnboxedIntArr`
data instance `Array Bool` = `BoolArr UnboxedBitVector`
data instance `Array (a, b)` = `PairArr (Array a) (Array b)`
data instance `Array (Array a)` = `ArrArr (Array Int) (Array a)`

Example

`Array (Int, Bool)`



Unboxed representation

- `[[:1, 2:], [::], [:3, 4, 5:]] ⇒ 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

Classic example

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

Classic example

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,

Elem [e] = *e* **for** *Collects [e]*

Elem BitSet = *Char* **for** *Collects BitSet*



With associated type synonym families

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

class *Collects* *c* **where**

type Elem *c*

-- definition varies with *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*

...



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 :: ???

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

$merge :: (Collects\ c1,\ Collects\ c2,\ ????)$
 $\Rightarrow c1 \rightarrow c2 \rightarrow c2$
 $merge\ c1\ c2 = foldr\ insert\ c2\ (toList\ c1)$

Make a collection from a list of elements

$fromList :: Collects\ c \Rightarrow [Elem\ c] \rightarrow c$
 $fromList\ l = foldr\ insert\ empty\ l$

Merge elements of one collection into another

$merge :: (Collects\ c1, Collects\ c2, Elem\ c1 \sim Elem\ c2)$
 $\Rightarrow c1 \rightarrow c2 \rightarrow c2$
 $merge\ c1\ c2 = foldr\ insert\ c2\ (toList\ c1)$

- We need **equality constraints**

Make a collection from a list of elements

$fromList :: Collects\ c \Rightarrow [Elem\ c] \rightarrow c$
 $fromList\ l = foldr\ insert\ empty\ l$

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

```
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             -- bounded lists as GADT
```

```
  BNil  :: BList Zero a
```

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

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

```
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             -- bounded lists as GADT
```

```
  BNil  :: BList Zero a
```

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

```
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

Type Families and Functional Dependencies

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. . .]



Implementation

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

Documentation

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