Generic Programming with Type Families
— Scrap Your Container Instances —

Manuel M. T. Chakravarty
University of New South Wales

Joint work with
Gabriele Keller
Roman Leshchinskiy
Simon Peyton Jones
Motivation

Non-parametric container types

- A parametric container type (such as `[e]`) uses a single representation and is independent of the type of elements.
Motivation

Non-parametric container types

- A parametric container type (such as \([e]\))
  - uses a single representation
  - independent of the type of elements.
- A non-parametric container type
  - changes its representation
  - in dependence on the type of elements it contains.
- This is usually for optimisation purposes.
- Examples of language features used for this purpose:
  - C++ (templates and traits)
  - Generic Haskell (type-indexed data types)
  - Haskell (type families)
Motivation

Non-parametric container types

- A parametric container type (such as [e])
  - uses a single representation
  - independent of the type of elements.
- A non-parametric container type
  - changes its representation
  - in dependence on the type of elements it contains.
- This is usually for optimisation purposes.
- Examples of language features used for this purpose:
  - C++ (templates and traits)
  - Generic Haskell (type-indexed data types)
  - Haskell (type families) \(\iff\) This Talk!

In the development version of the Glasgow Haskell Compiler (GHC).
Boxed array:

Array (Int, Bool)

Array example

data family Array e
Unboxed arrays:

\[ \text{ArrProd (ArrInt } u_a\text{) (ArrBool } b_v\text{) :: Array (Int, Bool)} \]

**Array example**

```
data family Array e  -- representation depends on e

data instance Array Int = ArrInt (UnbArray Int)
data instance Array Bool = ArrBool BitVector

data instance Array (e1, e2) = ArrProd (Array e1) (Array e2)
```
Problem solved?!?

- Data familes obviously do the job
- Thanks for listening to this very short talk!
### Problem solved?!?

- Data families obviously do the job
- Thanks for listening to this very short talk!

### The real problem solved in this talk

- Which type of elements can we store in such a non-parametric array?
- What about user-defined structures?
- Do we have to define a new set of array methods for every array element type?
Problem solved?!?

- Data families obviously do the job
- Thanks for listening to this very short talk!

The real problem solved in this talk

- Which type of elements can we store in such a non-parametric array?
- What about user-defined structures?
- Do we have to define a new set of array methods for every array element type?

The goal

- Data instances for `Array` only for limited set of element types: basic types, products & sums, and arrays(!)
- The same for array method implementations
- Derive implementation for user-defined algebraic types automatically
data MassPnt  -- mass points
data Tree     = Node MassPnt (Array Tree)

• Example application: Barnes-Hut algorithm to solve the $n$-body problem
• Tree performs spatial decomposition
• Levelwise representation ideal for parallel implementation
data MassPnt  -- mass points

data Tree       = Node MassPnt (Array Tree)

• Example application: Barnes-Hut algorithm to solve the $n$-body problem
• Tree performs spatial decomposition
• Levelwise representation ideal for parallel implementation
The Key Idea

All problems in computer science can be solved by another level of indirection.

— Butler Lampson
The Key Idea

All problems in computer science can be solved by another level of indirection.

— Butler Lampson

Two-level mapping

<table>
<thead>
<tr>
<th>element type</th>
<th>representation type</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>Repr</td>
</tr>
<tr>
<td>n : 1</td>
<td>r</td>
</tr>
</tbody>
</table>

1:1 ↘ Array

arr

- **Repr**: type synonym family maps unbound number of algebraic data types to finite number of representations
- **Array**: data type family mapping element representations to array representations
The Key Idea

All problems in computer science can be solved by another level of indirection.

— Butler Lampson

Two-level mapping

<table>
<thead>
<tr>
<th>element type</th>
<th>representation type</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>Repr</td>
</tr>
<tr>
<td>n : 1</td>
<td>r</td>
</tr>
<tr>
<td>1:1 Array</td>
<td>arr</td>
</tr>
</tbody>
</table>

- **Repr**: *type synonym family* maps unbound number of algebraic data types to finite number of representations
- **Array**: *data type family* mapping element representations to array representations

— We’ll look at this point first.
What are our representation types going to be?
What are our representation types going to be?

```haskell
data Int, ...  -- basic types
data Unit = Unit  -- singleton
data a :*: b = a :*: b  -- products
data a :+: b = Inl a | Inr b  -- sums
```
What are our representation types going to be?

data Int, ...  
data Unit = Unit  
data a :*: b = a :*: b  
data a :+: b = Inl a | Inr b  
data family Array r

– basic types
– singleton
– products
– sums
– arrays
What are our representation types going to be?

- basic types
- singleton
- products
- sums
- arrays

Type class categorising array elements

- data family as associated type

class ArrElem r where

data Array r

lengthA :: Array r -> Int
emptyA :: Array r
replicateA :: Int -> r -> Array r
(!:) :: Array r -> Int -> r

– and many more
Limited Set of Data Instances

What are our representation types going to be?

```hs
data Int, ...            -- basic types
data Unit = Unit         -- singleton
data a :*: b = a :*: b    -- products
data a :+: b = Inl a | Inr b -- sums
data family Array r      -- arrays
```

Class instances: basic types

```hs
instance ArrElem Int where
    data Array Int = ArrInt (UnbArray Int)
    lengthA (ArrInt ua) = lengthUA ua
    emptyA = ArrInt emptyUA
    replicateA n x = ArrInt (replicateUA n x)
    (ArrInt a) !: i = a!i
-- and so on
```

Manuel M. T. Chakravarty
Generic Programming with Type Families
What are our representation types going to be?

- basic types

```
data Int, ...                  -- basic types
```

- singleton

```
data Unit = Unit               -- singleton
```

- products

```
data a :+: b = Inl a | Inr b    -- products
```

- sums

```
data a :*: b = a :*: b          -- sums
```

- arrays

```
data family Array r           -- arrays
```

Class instances: products

```
instance (ArrElem r1, ArrElem r2) =>
  ArrElem (r1 :*: r2) where
  data Array (r1 :*: r2) = ArrProd (Array r1) (Array r2)
  lengthA (ArrProd arr1 _) = lengthA arr1
  emptyA                     = ArrProd emptyPA emptyPA
  replicateA n (x1 :*: x2) = ArrProd (replicatePA n x1)
                          (replicatePA n x2)
  (ArrProd a1 a2) !: i       = (a1!:i) :*: (a2!:i)
```
What are our representation types going to be?

- basic types
  
  ```haskell
  data Int, ...
  ```

- singleton
  
  ```haskell
  data Unit = Unit
  ```

- products
  
  ```haskell
  data a :*: b = a :*: b
  ```

- sums
  
  ```haskell
  data a :+: b = Inl a | Inr b
  ```

- arrays
  
  ```haskell
  data family Array r
  ```

Class instances: and so on

```
instance ArrElem Unit where ...
instance (ArrElem r1, ArrElem r2) =>
  ArrElem (r1 :+: r2) where ...
instance ArrElem r => ArrElem (Array r) where ...
```
Remember...

Two-level mapping

<table>
<thead>
<tr>
<th>element type</th>
<th>representation type</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>r</td>
</tr>
<tr>
<td>Repr</td>
<td>Array</td>
</tr>
<tr>
<td>$n : 1$</td>
<td>$1 : 1$</td>
</tr>
</tbody>
</table>

arr
An embedding projection

class Representable a where

  type Repr a  -- type function from a to representation of a
  toRepr :: a -> Repr a  -- embed
  fromRepr :: Repr a -> a  -- project

  type Repr a = a  -- default representation is the identity
  toRepr = id
  fromRepr = id
Generic Representation Types

An embedding projection

class Representable a where

  type Repr a         -- type function from a to representation of a
  toRepr :: a -> Repr a -- embed
  fromRepr :: Repr a -> a -- project

  type Repr a = a      -- default representation is the identity
  toRepr = id
  fromRepr = id

Arrays revisited

class ArrElem e where

  data Array e

  lengthA :: Array e -> Int
  emptyA :: Array e
  replicateA :: Int -> e -> Array e
  (!:)    :: Array e -> Int -> e
Generic Representation Types

An embedding projection

class Representable a where
  type Repr a  -- type function from a to representation of a
  toRepr :: a -> Repr a  -- embed
  fromRepr :: Repr a -> a  -- project

  type Repr a = a  -- default representation is the identity
  toRepr = id
  fromRepr = id

Arrays revisited

class Representable e => ArrElem e where
  data Array e  -- data family as associated type
    lengthA :: Array e -> Int
    emptyA :: Array e
    replicateA :: Int -> e -> Array e
    (!:) :: Array e -> Int -> e
Generic Representation Types

An embedding projection

class Representable a where

type Repr a  -- type function from a to representation of a

toRepr :: a -> Repr a  -- embed

fromRepr :: Repr a -> a  -- project

type Repr a = a  -- default representation is the identity

toRepr  = id

fromRepr = id

Default representations

instance Representable Int  -- and so on

instance Representable Unit

instance (Representable a, Representable b) =>
    Representable (a :*: b)

instance (Representable a, Representable b) =>
    Representable (a :+: b)

instance Representable e => Representable (PArray e)
**An embedding projection**

```haskell
class Representable a where
    type Repr a -- type function from a to representation of a
    toRepr :: a -> Repr a -- embed
    fromRepr :: Repr a -> a -- project

    type Repr a = a -- default representation is the identity
    toRepr = id
    fromRepr = id
```

**Example: spatial decomposition trees**

```haskell
data Tree = Node MassPnt (Array Tree)

instance Representable Tree where
    type Repr Tree = MassPnt :*: Array Tree
    toRepr (Tree mpnt ts) = mpnt :*: toRepr ts
    fromRepr (mpnt :*: ts) = Tree mpnt (fromRepr ts)
```
An embedding projection

class Representable a where

  type Repr a = a 
"-- default representation is the identity

toRepr :: a -> Repr a 
"-- embed

fromRepr :: Repr a -> a 
"-- project

Example: spatial decomposition trees

data Tree = Node MassPnt (Array Tree)

instance Representable Tree where

  type Repr Tree = MassPnt :*: Array Tree 

  toRepr (Tree mpnt ts) = mpnt :*: toRepr ts

  fromRepr (mpnt :*: ts) = Tree mpnt (fromRepr ts)
Example: spatial decomposition trees

data Tree = Node MassPnt (Array Tree)

instance Representable Tree where
  type Repr Tree = MassPnt :*: Array Tree
  toRepr (Tree mpnt ts) = mpnt :*: toRepr ts
  fromRepr (mpnt :*: ts) = Tree mpnt (fromRepr ts)
Example: spatial decomposition trees

data Tree = Node MassPnt (Array Tree)

instance Representable Tree where
    type Repr Tree = MassPnt :*: Array Tree
    toRepr (Tree mpnt ts) = mpnt :*: toRepr ts
    fromRepr (mpnt :*: ts) = Tree mpnt (fromRepr ts)

Stub instance

instance ArrElem Tree where
    newtype Array Tree = ArrTree (Array (Repr Tree))
    lengthA (ArrTree a) = lengthA a
    emptyA = ArrTree emptyA
    replicateA n e = ArrTree (replicateA n (toRepr e))
    (ArrTree a) !: i = fromRepr (a!(:i))
Beyond the Basic Idea

Other choice of representation types possible

- For example, we could use $n$-ary products and sums
Beyond the Basic Idea

Other choice of representation types possible

- For example, we could use $n$-ary products and sums

Eliminate the stub instances

- Stubs define identities on the type and value level
- See `PArray` of Data Parallel Haskell
Beyond the Basic Idea

Other choice of representation types possible

- For example, we could use $n$-ary products and sums

Eliminate the stub instances

- Stubs define identities on the type and value level
- See `PArray` of Data Parallel Haskell

Compiler support: deriving `Representable`

data Tree = Node MassPnt (Array Tree)

instance Representable Tree where
  type Repr Tree = MassPnt :*: Array Tree
  toRepr (Tree mpnt ts) = mpnt :*: toRepr ts
  fromRepr (mpnt :*: ts) = Tree mpnt (fromRepr ts)
Beyond the Basic Idea

Other choice of representation types possible
- For example, we could use $n$-ary products and sums

Eliminate the stub instances
- Stubs define identities on the type and value level
- See PArray of Data Parallel Haskell

Compiler support: deriving Representable

```
data Tree = Node MassPnt (Array Tree)

deriving Representable
```
Conclusions

Non-parametric containers

- Goal: Avoid proliferation of container instances
- Key ideas:
  1. Map user-defined types to fixed set of representation types
  2. Type synonym family makes the mapping transparent
- A little compiler support makes it even more convenient

Related work

- C++: unclear how to prevent tedious definition of instances for user-defined types
- Generic Haskell: special language extension and compiler; fixed set of representation types
Conclusions

Non-parametric containers

- Goal: Avoid proliferation of container instances
- Key ideas:
  1. map user-defined types to fixed set of representation types
  2. type synonym family makes the mapping transparent
- A little compiler support makes it even more convenient

Related work

- C++: unclear how to prevent tedious definition of instances for user-defined types
- Generic Haskell: special language extension and compiler; fixed set of representation types
-- Product-sum type representations
-- --------------------------------

data Unit = Unit
  deriving Show

data (:+:) a b = Inl a | Inr b
  deriving Show

data (:*:) a b = a :*: b
  deriving Show

class Representable a where
  type Repr a
  toRepr :: a -> Repr a
  fromRepr :: Repr a -> a

instance Representable Unit where
  type Repr Unit = Unit
  toRepr = id
  fromRepr = id

instance Representable () where
  type Repr () = Unit
  toRepr () = Unit
  fromRepr Unit = ()

instance Representable Int where
  type Repr Int = Int
  toRepr = id
  fromRepr = id

instance Representable Char where
  type Repr Char = Char
  toRepr = id
  fromRepr = id

instance (Representable a, Representable b) =>
  Representable (a :*: b) where
  type Repr (a :*: b) = a :*: b
  toRepr = id
  fromRepr = id

instance (Representable a, Representable b) =>
  Representable (a, b) where
  type Repr (a, b) = a :*: b
  toRepr (x, y) = x :*: y
  fromRepr (x :*: y) = (x, y)

instance (Representable a, Representable b) =>
  Representable (a :+: b) where
  type Repr (a :+: b) = a :+: b
  toRepr = id
  fromRepr = id

instance (Representable a, Representable b) =>
  Representable (Either a b) where
  type Repr (Either a b) = a :+: b
  toRepr (Left x) = Inl x
  toRepr (Right y) = Inr y
  fromRepr (Inl x) = Left x
  fromRepr (Inr x) = Right x
class Representable k => GMapKey k where
  data GMap k :: * -> *
  empty :: GMap k v
  lookup :: k -> GMap k v -> Maybe v
  insert :: k -> v -> GMap k v -> GMap k v

instance GMapKey Int where
  data GMap Int v = GMapInt (Map.Map Int v)
  empty = GMapInt Map.empty
  lookup k (GMapInt m)
    = lookup k m
  insert k v (GMapInt m)
    = GMapInt (Map.insert k v m)

instance GMapKey Char where
  data GMap Char v = GMapChar (GMap Int v)
  empty = GMapChar empty
  lookup k (GMapChar m)
    = lookup (ord k) m
  insert k v (GMapChar m)
    = GMapChar (insert (ord k) v m)

instance GMapKey Unit where
  data GMap Unit v = GMapUnit (Maybe v)
  empty = GMapUnit Nothing
  lookup Unit (GMapUnit v)
    = v
  insert Unit v (GMapUnit _)
    = GMapUnit $ Just v

instance (GMapKey a, GMapKey b) =>
  GMapKey (a :*: b) where
  data GMap (a :*: b) v = GMapPair (GMap a (GMap b v))
  empty = GMapPair empty
  lookup (a :*: b) (GMapPair gm)
    = lookup a gm >>= lookup b
  insert (a :*: b) v (GMapPair gm)
    = GMapPair $ case lookup a gm of
      Nothing -> insert a (insert b v empty) gm
      Just gm2 -> insert a (insert b v gm2) gm

instance (GMapKey a, GMapKey b) =>
  GMapKey (a :+: b) where
  data GMap (a :+: b) v = GMapEither (GMap a v) (GMap b v)
  empty = GMapEither empty empty
  lookup (Inl a) (GMapEither gm1 _gm2)
    = lookup a gm1
  lookup (Inr b) (GMapEither _gm1 gm2)
    = lookup b gm2
  insert (Inl a) v (GMapEither gm1 gm2)
    = GMapEither (insert a v gm1) gm2
  insert (Inr a) v (GMapEither gm1 gm2)
    = GMapEither gm1 (insert a v gm2)
-- Derived instances
-- -----------------

-- Generic list representation
--
-- * Could be added to Typeable
-- * Could use n-ary sums and products
--
instance Representable [a] where
   type Repr [a] = Unit :+: (a :*: [a])
   toRepr []    = Inl Unit
   toRepr (x:xs) = Inr (x :*: xs)
   fromRepr (Inl Unit) = []
   fromRepr (Inr (x :*: xs)) = x:xs

-- List-indexed maps
--
-- * Could also be generated
--
instance GMapKey a => GMapKey [a] where
   newtype GMap [a] v = GMapList (GMap (Repr [a]) v)
   empty           = GMapList empty
   lookup k (GMapList gm) = lookup (toRepr k) gm
   insert k v (GMapList gm) = GMapList $ insert (toRepr k) v gm