Universiteit Utrecht

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

Advanced Functional Programming

2011-2012, period 2

Andres Löh and Doaitse Swierstra

Department of Information and Computing Sciences
Utrecht University

Jan 10, 2012

12. Functional Dependencies, Generalized Algebraic Datatypes (GADTs), The Lambda Cube

This lecture

12.1 Multiple parameters and functional dependencies

Multi-parameter type classes

This extension allows type classes to have multiple parameters:

```
class Collection c a where
   union :: c a → c a → c a
   elem  :: a → c a → Bool
   empty :: c a
```

Multi-parameter type classes

This extension allows type classes to have multiple parameters:

```
class Collection c a where
  union  :: c a → c a → c a
  elem   :: a → c a → Bool
  empty :: c a
```

Why is

```
class Collection c where
  union  :: c a → c a → c a
  elem   :: a → c a → Bool
  empty :: c a
```

not an option?

Multi-parameter type classes (contd.)

This form is still suboptimal:

```
class Collection c a where
  union :: c a → c a → c a
  elem  :: a   → c a → Bool
  empty ::             c a
```

What about Data.IntSet.IntSet? It is not of the form c a, so it cannot be made an instance of Collection, even though it supports all the methods.

Multi-parameter type classes (contd.)

This form is still suboptimal:

```
class Collection c a where
  union  :: c a → c a → c a
  elem   :: a   → c a → Bool
  empty  ::             c a
```

What about Data.IntSet.IntSet? It is not of the form c a, so it cannot be made an instance of Collection, even though it supports all the methods.

Another idea:

```
class Collection ca a where
  union  :: ca → ca → ca
  elem   :: a  → ca → Bool
  empty  ::           ca
```

Multi-parameter type classes (contd.)

```
class Collection ca a where
  union :: ca → ca → ca
  elem  :: a → ca → Bool
  empty :: ca
```

Problem 1

```
empty :: (Collection ca a) ⇒ ca
```

has an ambiguous type.

Problem 2

```
test :: (Collection ca Bool, Collection ca String) ⇒ ca → Bool
test coll = elem True coll ∧ elem "foo" coll
```

is type-correct, but intuitively should not be.

Functional dependencies

```
class Collection ca a | ca → a where
  ...
```

- This indicates that ca determines a. It restricts the admissible instances.

  ```
  instance Collection IntSet Int
  ```

  is possible, a subsequent

  ```
  instance Collection IntSet Bool
  ```

  is now disallowed.
- Solves both the problems just mentioned ...

Functional dependencies (contd.)

With functional dependencies, the type

```
empty :: (Collection ca a) ⇒ ca
```

is no longer ambiguous.

Functional dependencies (contd.)

With functional dependencies, the type

```
empty :: (Collection ca a) ⇒ ca
```

is no longer ambiguous.

```
instance Collection IntSet Int
empty :: IntSet
```

Now correct. The inferred class constraint Collection IntSet a can be improved to Collection IntSet Int and then be reduced.

Functional dependencies (contd.)

```
test :: (Collection ca Bool, Collection ca String) ⇒ ca → Bool
test coll = elem True coll ∧ elem "foo" coll
```

No longer ok, because the two constraints cannot be satisfied at the same time while respecting the functional dependency.

Functional dependencies (contd.)

Functional dependencies are extremely powerful and (in conjunction with other extensions) can encode many computations:

```
data Zero = Zero
data Succ a = Succ a
class Add x y z | x y → z where
  add :: x → y → z
instance Add Zero x x
  where add Zero x     = x
instance Add n x r ⇒ Add (Succ n) x (Succ r)
  where add (Succ n) x = Succ (add n x)
```

```
Main⟩ :t add (Succ Zero) (Succ Zero)
add (Succ Zero) (Succ Zero) :: Succ (Succ Zero)
```

Addition performed by the type system!

12.2 Type families

Associated types

An alternative to functional dependencies. Type synonyms and datatypes are allowed in classes:

```
class Collection c where
   type Elem c
   union  :: c → c → c
   elem   :: Elem c → c → Bool
   empty :: c
instance Collection IntSet where
   type Elem IntSet = Int
   ...
```

Associated type synonyms trigger equality constraints, a different form of qualified types:

```
elem False :: (Bool∼Elem c, Collection c) ⇒ c → Bool
```

Type families

Like associated types, but the class declaration remains implicit:

```
type family Elem c :: *
type instance Elem IntSet = Int
```

Associated datatypes and datatype families are also supported.

Type families (contd.)

Using type families, type-level functions look a bit more like ordinary functions:

```
type family     Add n x :: *
type instance Add Zero       x = x
type instance Add (Succ n) x = Succ (Add n x)
```

Fundeps vs. type families

Functional dependencies are controversial, because

- they lead to logic programming on the type level (as opposed to functional programming),
- their interaction with other type system features (such as GADTs) is somewhat broken,
- because their use has some strange restrictions.

The latter features are problems with the implementation rather than the concepts.

Fundeps vs. type families (contd.)

Type families have been proposed as a replacement for functional dependencies.

- Type families allow a more functional style of programming.
- However, they expose a new language concept to the user (equality constraints).
- Just those equality constraints make the connection to GADTs somewhat easier.
- They are much more recent, therefore most libraries (monad transformers, HList, . . .) still use functional dependencies.

Case study: Heterogeneous lists

The HList library makes use of functional dependencies in order to support **heterogenous lists**.

```
data HNil       = HNil
data HCons e l  = HCons e l
type (:∗:)      = HCons
class HMap f l l′ | f l → l′ where hMap :: f → l → l′
instance HMap f HNil HNil where
   hMap f HNil           = HNil
instance (Apply f x y, HMap f xs ys) ⇒
            HMap f (HCons x xs) (HCons y ys) where
   hmap f  (HCons x xs) = HCons (apply f x) (hmap f xs)
class Apply f a r | f a → r where apply :: f → a → r
instance Apply (x → y) x y
```

Heterogeneous lists (contd.)

The HList library can be used to encode

- typed heterogenous lists or stacks
- extensible records
- objects

More class system extensions . . .

- Local or named instances.
- Implicit parameters.
- Explicit implicit parameters.
- Quantified instances.
- Recursive dictionaries.
- Alternative translation methods.
- Cyclic class hierarchy.
- Backtracking.
- . . .

12.3 GADTs

A datatype

```
data Tree a = Leaf
            | Node (Tree a) a (Tree a)
```

Introduces:

A datatype

```
data Tree a = Leaf
            | Node (Tree a) a (Tree a)
```

Introduces:

- a new datatype Tree of kind $* \to *$.
- constructor functions

```
Leaf :: Tree a
Node :: Tree a → a → Tree a → Tree a
```

- the possiblity to use the constructors Leaf and Node in patterns.

Universiteit Utrecht

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

Alternative syntax

Observation

The types of the constructor functions contain sufficient information to describe the datatype.

```
data Tree :: * → * where
  Leaf  :: Tree a
  Node :: Tree a → a → Tree a → Tree a
```

Are there any restrictions regarding the types of the constructors?

Algebraic datatypes

Constructors of an algebraic datatype T must:

- target type T,
- result in a simple type, i.e., $\mathsf{T}\ \mathsf{a}_1 \ldots \mathsf{a}_\mathsf{n}$ where $\mathsf{a}_1, \ldots, \mathsf{a}_\mathsf{n}$ are distinct type variables.

Question

Does it make sense to lift these restrictions?

Excursion: Writing an interpreter

```
data Expr =
      Int    Int
    | Bool   Bool
    | IsZero Expr
    | Plus   Expr Expr
    | If     Expr Expr Expr
```

```
data Expr :: * where
   Int    :: Int → Expr
   Bool   :: Bool → Expr
   IsZero :: Expr → Expr
   Plus   :: Expr → Expr → Expr
   If     :: Expr → Expr → Expr → Expr
```

Imagined concrete syntax:

```
if isZero (0 + 1) then False else True
```

Abstract syntax:

```
If (IsZero (Plus (Int 0) (Int 1))) (Bool False) (Bool True)
```

Evaluation

```
data Val =
      VInt   Int
    | VBool Bool
```

```
data Val :: * where
   VInt   :: Int → Val
   VBool :: Bool → Val
```

```
eval :: Expr → Val
eval (Int n)        = VInt n
eval (Bool b)       = VBool b
eval (IsZero e)     = case eval e of
                          VInt n → VBool (n == 0)
                          _      → error "type error"
eval (Plus e₁ e₂) = case (eval e₁, eval e₂) of
                          (VInt n1, VInt n2) → VInt (n1 + n2)
                          _                  → error "type error"
eval (If e₁ e₂ e₃) = case eval e₁ of
                          VBool b → if b then eval e₂ else eval e₃
                          _       → error "type error"
```

Evaluation (contd.)

- Evaluation code is mixed with code for handling type errors.
- The evaluator uses tags (i.e., constructors) to dinstinguish values – these tags are maintained and checked at run time.

Evaluation (contd.)

- Evaluation code is mixed with code for handling type errors.
- The evaluator uses tags (i.e., constructors) to dinstinguish values – these tags are maintained and checked at run time.
- Run-time type errors can, of course, be prevented by writing a type checker.
- But even if we know that we only have type-correct terms, the Haskell compiler does not enforce this.

An idea

What if we encode the type of the term in the Haskell type?

```
data Expr :: * where
  Int    :: Int → Expr
  Bool   :: Bool → Expr
  IsZero :: Expr → Expr
  Plus   :: Expr → Expr → Expr
  If     :: Expr → Expr → Expr → Expr
data Expr :: * → * where
  Int    :: Int → Expr Int
  Bool   :: Bool → Expr Bool
  IsZero :: Expr Int → Expr Bool
  Plus   :: Expr Int → Expr Int → Expr Int
  If     :: Expr Bool → Expr a → Expr a → Expr a
```

GADTs

GADTs lift the restriction that constructors must target a simple type.

- Constructors can target a subset of the type.
- Interesting consequences for pattern matching:
  - when case-analyzing an Expr Int, it cannot be constructed by Bool or IsZero;
  - when case-analyzing an Expr Bool, it cannot be constructed by Int or Plus;
  - when case-analyzing an Expr a, once we encounter the constructor IsZero in a pattern, we know that we have in fact a Expr Bool;
  - ...

Evaluation revisited

```
eval :: Expr a → a
eval (Int n)        = n
eval (Bool b)       = b
eval (IsZero e)     = (eval e) == 0
eval (Plus e₁ e₂)   = eval e₁ + eval e₂
eval (If e₁ e₂ e₃)  = if eval e₁ then eval e₂ else eval e₃
```

- No possibility for run-time failure (modulo $\perp$).
- No tags required.
- Pattern matching on a GADT requires a type signature. Why?

Type signatures are required . . .

```
data X :: * → * where
    C :: Int → X Int
    D :: X a
f (C n) = [n]
f D     = []
```

Question

What is the type of f?

Universiteit Utrecht

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

Type signatures are required . . .

```
data X :: * → * where
    C :: Int → X Int
    D :: X a
f (C n) = [n]
f D     = []
```

Question

What is the type of f?

Answer

```
f :: X a → [Int]
f :: X a → [a]
```

None of the two is an instance of the other.

GADTs subsume existentials

Let us extend the expression types with pair construction and projection:

```
data Expr :: * → * where
  Int    :: Int → Expr Int
  Bool   :: Bool → Expr Bool
  IsZero :: Expr Int → Expr Bool
  Plus   :: Expr Int → Expr Int → Expr Int
  If     :: Expr Bool → Expr a → Expr a → Expr a
  Pair   :: Expr a → Expr b → Expr (a, b)
  Fst    :: Expr (a, b) → Expr a
  Snd    :: Expr (a, b) → Expr b
```

For Fst and Snd, the type of the non-projected component is hidden.

Evaluation again

```
eval :: Expr a → a
eval ...
eval (Pair x y) = (eval x, eval y)
eval (Fst p)    = fst (eval p)
eval (Snd p)    = snd (eval p)
```

12.4 Example: Vectors

Natural numbers and vectors

Natural numbers can be encoded as types – no constructors are required.

```
data Zero
data Succ a
```

Natural numbers and vectors

Natural numbers can be encoded as types – no constructors are required.

```
data Zero
data Succ a
```

Vectors are lists with a fixed number of elements:

```
data Vec :: * → * → * where
   Nil  :: Vec a Zero
   Cons :: a → Vec a n → Vec a (Succ n)
```

Unlike HLists, vectors are homogeneous.

Type-safe head and tail

```
head :: Vec a (Succ n) → a
head (Cons x xs) = x
tail :: Vec a (Succ n) → Vec a n
tail (Cons x xs)   = xs
```

- No case for Nil is required.
- Actually, a case for Nil results in a type error.

More functions on vectors

```
map :: (a → b) → Vec a n → Vec b n
map f Nil          = Nil
map f (Cons x xs) = Cons (f x) (map f xs)
zipWith :: (a → b → c) → Vec a n → Vec b n → Vec c n
zipWith op Nil          Nil          = Nil
zipWith op (Cons x xs) (Cons y ys) = Cons (op x y)
                                          (zipWith op xs ys)
```

We require that the two vectors have the same length!

Yet more functions on vectors

```
snoc :: Vec a n → a → Vec a (Succ n)
snoc Nil          y  = Cons y Nil
snoc (Cons x xs) y  = Cons x (snoc xs y)
reverse :: Vec a n → Vec a n
reverse Nil          = Nil
reverse (Cons x xs) = snoc xs x
```

What about (++)?

Universiteit Utrecht

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

12.5 Problematic functions

Problematic functions

Append $(+\!\!+)$:

```
(++) :: Vec a m → Vec a n → Vec a (Sum m n)
```

Do we need functions on the type level?

Converting from lists to vectors:

```
fromList :: [a] → Vec a n
```

Where does n come from?

Writing vector append

There are multiple options to solve that problem:

- construct explicit evidence,
- use a type family.

Explicit evidence

We encode the addition as another GADT:

```
data Sum :: * → * → * → * where
   SumZero ::                   Sum Zero     n n
   SumSucc :: Sum m n s → Sum (Succ m) n (Succ s)
append :: Sum m n s → Vec a m → Vec a n → Vec a s
append SumZero      Nil         ys = ys
append (SumSucc p) (Cons x xs) ys = Cons x (append p xs ys)
```

Disadvantage: we must construct the evidence by hand!

Explicit evidence

We encode the addition as another GADT:

```
data Sum :: * → * → * → * where
  SumZero ::                    Sum Zero     n n
  SumSucc :: Sum m n s → Sum (Succ m) n (Succ s)
append :: Sum m n s → Vec a m → Vec a n → Vec a s
append SumZero      Nil         ys = ys
append (SumSucc p) (Cons x xs) ys = Cons x (append p xs ys)
```

Disadvantage: we must construct the evidence by hand!

We could use a multi-parameter type class with functional dependencies, but even better is a ...

Type family

```
type family   Sum m        n :: *
type instance Sum Zero     n = n
type instance Sum (Succ m) n = Succ (Sum m n)

(++) :: Vec a m → Vec a n → Vec a (Sum m n)
Nil       ++  ys = ys
Cons x xs ++  ys = Cons x (xs ++ ys)
```

Converting between lists and vectors

Unproblematic:

```
toList :: Vec a n → [a]
toList Nil          = []
toList (Cons x xs) = x : toList xs
```

Does not work:

```
fromList :: [a] → Vec a n
fromList []       = Nil
fromList (x : xs) = Cons x (fromList xs)
```

Why? The type says that the result must be polymorphic in n, and it is not!

From lists to vectors

We can

- specify the length,
- hide the length using an existential type.

For the former, we have to reflect type-level natural numbers on the value level:

```
data Nat :: * → * where
   Zero :: Nat Zero
   Succ :: Nat n → Nat (Succ n)
```

From lists to vectors (contd.)

```
data Nat :: * → * where
   Zero :: Nat Zero
   Succ :: Nat n → Nat (Succ n)
fromList :: Nat → [a] → Vec a n
fromList Zero     []     = Nil
fromList (Succ n) (x:xs) = Cons x (fromList n xs)
fromList _        _      = error "wrong length!"
```

We have to know the length in advance.

From lists to vectors (contd.)

Using an existential type (in GADT notation):

```
data VecAny :: * → * where
  VecAny :: Vec a n → VecAny a
fromList :: [a] → VecAny a
fromList []       = VecAny Nil
fromList (x : xs) = case fromList xs of
                      VecAny ys → VecAny (Cons x ys)
```

We can combine the ideas and include a Nat in the packed type:

```
data VecAny :: * → * where
  VecAny :: Nat n → Vec a n → VecAny a
```