IA014: Advanced Functional Programming

8. GADT – Generalized Algebraic Data Types (and type extensions)

Jan Obdržálek obdrzalek@fi.muni.cz

Faculty of Informatics, Masaryk University, Brno

Motivation

Algebraic Data Types (ADT)

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

What do we get?

- type constructor `Tree`, of kind `* -> *`
- two value constructors `Leaf` and `Node`, of types

```
> :t Leaf
Leaf :: Tree a
> :t Node
Node :: Tree a -> a -> Tree a -> Tree a
```

- value constructors can be used for pattern matching

Alternative syntax

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

Observation: The types of value constructors fully describe the datatype.

```
data Tree a where
    Leaf :: Tree a
    Node :: Tree a -> a -> Tree a -> Tree a
```

another example

```
data Either a b = Left a | Right b
```

can be written as

```
data Either a b where
    Left  :: a -> Either a b
    Right :: b -> Either a b
```

Range restriction of ADTs

```
data Tree a where
    Leaf :: Tree a
    Node :: Tree a -> a -> Tree a -> Tree a
```

- both constructors target `Tree a`

```
data Either a b where
    Left  :: a -> Either a b
    Right :: b -> Either a b
```

- both constructors target `Either a b`

In ADTs, all constructors have *identical range types*.

Can this restriction be lifted?

Example: Expression evaluator

```
data Expr = I Int
          | AddInt Expr Expr

eval :: Expr -> Int
eval e = case e of
    I i -> i
    AddInt e1 e2 -> eval e1 + eval e2
```

adding Booleans

```
data Expr = I Int
          | B Bool
          | AddInt Expr Expr
          | IsZero Expr

eval e = ...
    B b -> b
    IsZero e -> (eval e) == 0
```

What is the type signature of `eval`?

Multi-type evaluator

What is the type of `eval`?

```
> :t IsZero (B True)
IsZero (B True) :: Expr
> :t AddInt (I 5) (B True)
AddInt (I 5) (B True) :: Expr
```

- possible results: `Int`, `Bool`, failure
- solution: `Maybe (Either Int Bool)`

```
eval :: Expr -> Maybe (Either Int Bool)

eval e = case e of
    AddInt e1 e2 -> case (eval e1, eval e2) of
        (Just (Left i1), Just (Left i2)) -> Just $ Left $ i1 + i2
        _ -> fail "AddInt takes two integers"
    ...
```

Problem: unreadable, exhaustive and clumsy

Phantom types

Example: `newtype Const a b = Const getConst :: a`

A *phantom type* is a parametrised type whose parameters do not all appear on the right-hand side of its definition.

```
data Expr t = I Int
            | B Bool
            | AddInt (Expr Int) (Expr Int)
            | IsZero (Expr Int)
```

However:

```
> :t IsZero (B True)
IsZero (B True) :: Expr t
```

Malformed expressions still typecheck!

Explicitly typing constructors

```
> :t IsZero (B True)
IsZero (B True) :: Expr t
```

We need to provide the type information explicitly:

```
i :: Int -> Expr Int
i = I
b :: Bool -> Expr Bool
b = B
isZero :: Expr Int -> Expr Bool
isZero = IsZero
```

Typechecker now rejects malformed expressions:

```
> :t isZero (b True)
Couldn't match expected type `Int' with actual type `Bool'...
> :t isZero (i 5)
isZero (i 5) :: Expr Bool
```

GADT evaluator 1/2

We can now put everything together:

1. define the type and its constructors:
   (giving explicit types)

```
data Expr t where
    I       :: Int -> Expr Int
    B       :: Bool -> Expr Bool
    AddInt  :: Expr Int -> Expr Int -> Expr Int
    IsZero  :: Expr Int -> Expr Bool
    If      :: Expr Bool -> Expr t -> Expr t -> Expr t
```

2. malformed expressions are now rejected by the typechecker:

```
> :t IsZero (B True)
Couldn't match expected type `Int' with actual type `Bool'...
> :t IsZero (I 5)
IsZero (I 5) :: Expr Bool
```

GADT evaluator 2/2

3. the evaluator itself is almost trivial:

```
eval :: Expr t -> t
eval (I i)          = i
eval (B b)          = b
eval (AddInt e1 e2) = eval e1 + eval e2
eval (IsZero e)     = eval e == 0
eval (If e1 e2 e3)  = if eval e1
                        then eval e2
                        else eval e3
```

GADTs

- unlike for algebraic datatypes, constructors can target only a *subset* of the type:

```
data Expr t where
    I :: Int  -> Expr Int
    B :: Bool -> Expr Bool
```

- pattern matching causes type refinement:

```
eval :: Expr t -> t
eval (I i) = ...
```

  (the type `t` is refined to `Int`)

- type refinement is carried out based on *user supplied type annotations*

Existential types as GADTs

Existential types drop the restriction that every type variable that appears on the right-hand side must also appear on the left-hand side (when creating a new type)

- example: `show`able heterogenous list

```
data Obj = forall a. (Show a) => Obj a

xs :: [Obj]
xs = [Obj 1, Obj "foo", Obj 'c']

doShow :: [Obj] -> String
doShow [] = ""
doShow ((Obj x):xs) = show x ++ doShow xs
```

- subsumed by GADTs:

```
data Obj where
   Obj :: Show a => a -> Obj
```

Type inference for GADTs

Is it necessary to give all those type signatures?

```
data Test t where
    TInt :: Int -> Test Int
    TString :: String -> Test String

f (TString s) = s
```

- two possible principal types:

  ```
  f :: Test t -> [Char]
  f :: Test t -> t
  ```

- neither one is an instance of the other
- *HM always derives the principal (most general) type!*

This fails to typecheck:

```
f' (TString s) = s
f' (TInt i) = i
```

Adding `f' :: Test t -> t` fixes the problem.

Safe Lists and Vectors

Revision: Standard lists

```
data List t = Nil | Cons t (List t)
```

In the GADT syntax:

```
data List t where
    Nil :: List t
    Cons :: t -> List t -> List t
```

The `head` function is defined as:

```
listHead :: List t -> t
listHead (Cons a _) = a
listHead Nil = error "list is empty"
```

Problem?

```
> :t listHead Nil
listHead Nil :: t
> listHead Nil
*** Exception: list is empty
```

Can fail at runtime.

Safe lists

A list can be either empty, or non-empty:

```
data Empty
data NonEmpty
```

We remember this information in the type of the list:

```
data SafeList a b where
     Nil :: SafeList a Empty
     Cons:: a -> SafeList a b -> SafeList a NonEmpty

safeHead :: SafeList a NonEmpty -> a
safeHead (Cons x _) = x
```

`safeHead` is safe:

```
> safeHead (Cons "hiya" Nil)
"hiya"
> safeHead Nil

<interactive>:1:9:
    Couldn't match `NonEmpty' against `Empty' ...
```

Vectors – Lists of a fixed length

To express list length, we need to encode natural numbers on the type level:

```
data Zero
data Succ n
```

Vectors – lists with a fixed number of elements:

```
data Vec a n where
    Nil :: Vec a Zero
    Cons :: a -> Vec a n -> Vec a (Succ n)
```

Example:

```
> :t Cons 'a' (Cons 'b' Nil)
Cons 'a' (Cons 'b' Nil) :: Vec Char (Succ (Succ Zero))
> :t Nil
Nil :: Vec a Zero
```

Safe head function

```
headSafe :: Vec a (Succ n) -> a
headSafe (Cons x _) = x
```

Note: no case for `Nil` required.

- `:t headSafe Nil` fails

```
> :t headSafe Nil
    Couldn't match type `Zero' with `Succ n0' ...
```

- adding a case: `headSafe Nil = error "bad, bad boy"`

```
> :l gadt.hs
[1 of 1] Compiling Main             ( gadt.hs, interpreted )

gadt.hs:43:10:
    Couldn't match type `Succ n' with `Zero'
    Inaccessible code in
      a pattern with constructor
        Nil :: forall a. Vec a Zero,
      in an equation for `headSafe'
```

More functions on vectors

```
mapSafe :: (a -> b) -> Vec a n -> Vec b n
mapSafe _ Nil = Nil
mapSafe f (Cons x xs) = Cons (f x) (mapSafe f xs)

zipWithSafe :: (a -> b -> c) -> Vec a n -> Vec b n -> Vec c n
zipWithSafe f Nil Nil = Nil
zipWithSafe f (Cons x xs) (Cons y ys) =
                  Cons (f x y) (zipWithSafe f xs ys)
```

In both cases above, it is necessary to explicitly declare the type signature!

Even more functions on vectors

Reversing vectors is also possible:

```
snoc :: Vec a n -> a -> Vec a (Succ n) -- necessary
snoc Nil         y = Cons y Nil
snoc (Cons x xs) y = Cons x (snoc xs y)

reverseSafe :: Vec a n -> Vec a n      -- necessary
reverseSafe Nil = Nil
reverseSafe (Cons x xs) = snoc xs x
```

Problematic case:
What if we want to join two vectors?

```
append :: Vec a n -> Vec a m -> Vec a (m+n)
```

The problem: `m` *and* `n` *are types!*

Vector join: solution 1

We encode the addition as (yet another) GADT:

```
data Sum m n s 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)
```

- you essentially provide a “proof” how long the first vector is

```
> append (SumSucc(SumSucc SumZero)) (Cons 'a' (Cons 'b' Nil))
       (Cons 'c' Nil)
```

- this evidence is constructed by hand
- not exactly practical

Type families

- *indexed type family* is a partial function at a type level
- parameters of this function are called *indices*
- unlike type constructors, this function does not have to be defined for all types
- example:

```
data family T a :: *
data instance T Int  = T1 Int | T2 Bool
data instance T Char = TC Bool
```

- `a` is the index here
- the kind signature of `T` is `* -> *` and can be omitted
- value constructors can be completely different
  (unlike for parameterized types)

Vector join: solution 2

- using *type families*
- HASKELL: pragma `{-# LANGUAGE TypeFamilies #-}`

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

append2 :: Vec a m -> Vec a n -> Vec a (SSum m n)
append2 Nil ys = ys
append2 (Cons x xs) ys = Cons x (append2 xs ys)
```

- note that `type instance` defines new type synonyms, not new types (unlike `data instance`)

More kinds

The standard kind system (using only `*` and `->`) is too limited. E.g the following does not make sense, but still typechecks:

```
> :k Vec Bool Bool
Vec Bool Bool :: *
```

The problem here is that `Vec :: * -> * -> *`.

DataKinds

- `{-# LANGUAGE DataKinds #-}`
- `Nat` gets *promoted* to a new kind
- now `Vec :: * -> Nat -> *`

```
> :k Vec Bool Bool

<interactive>:1:10:
    Kind mis-match
    The second argument of `Vec' should have kind `Nat',
    but `Bool' has kind `*'
    In a type in a GHCi command: Vec Bool Bool
```

Data type promotion

- complements kind polymorphism
- value constructors *also* become type constructors

	Original data type and example value	Promoted kind and example type
kinds		'Nat
types	Nat	'Succ ('Succ 'Zero)
values	Succ (Succ Zero)	

  - quote is used to resolve ambiguity (e.g. `'Zero` is a type)

```
> type T2 = Succ (Succ Zero)
> :i T2
type T2 = 'Succ ('Succ 'Zero)
> let a = Succ (Succ Zero)
> :t a
a :: Nat
```

Vectors can be tricky ...

Consider a function which produces a list of `n` values `a`:

```
repeat1 :: Nat -> a -> Vec a n
repeat1 Zero     _ = Nil
repeat1 (Succ n) a = Cons a (repeat1 n a)
```

This fails to typecheck:
There is no relation between the first argument and n

Singleton types

- types containing *only one value*
- example: Peano numbers:

```
data SNat n where
  SZ :: SNat 'Zero
  SS :: SNat n -> SNat ('Succ n)
```

- every *type* of kind `Nat` corresponds to exactly one value
- kind `Nat` is *mirrored* in the constructors of `SNat`
- `repeat` can now easily be written as:

```
repeat2 :: SNat n -> a -> Vec a n
repeat2 SZ     _ = Nil
repeat2 (SS n) a = Cons a (repeat2 n a)
```

Another example: 2-3 trees

- Every non-leaf is a 2-node or a 3-node.
  - A 2-node contains one data item and has two children.
  - A 3-node contains two data items and has 3 children.
- All leaves are at the same level (the bottom level)
- Every leaf node will contain 1 or 2 fields.

```
data Node s a where
    Leaf2 :: a -> Node Zero a
    Leaf3 :: a -> a -> Node Zero a
    Node2 :: Node s a -> a -> Node s a -> Node (Succ s) a
    Node3 :: Node s a -> a -> Node s a -> a -> Node s a -> Node (Succ

data BTree a where
    Root0 :: BTree a
    Root1 :: a -> BTree a
    RootN :: Node s a -> BTree a
```

Type system guarantees that:

- only balanced trees can be constructed
- i.e. operations like `insert` do not break this property

Going GADTless

GADTless evaluator 1/3

```haskell
class Expr e where
    intVal  :: Int -> e Int
    boolVal :: Bool -> e Bool
    add     :: e Int -> e Int -> e Int
    isZero  :: e Int -> e Bool
    if'     :: e Bool -> e t -> e t -> e t
```

Typechecker in this case also rejects malformed expressions:

```
> :t isZero $ boolVal True
   Couldn't match type `Bool' with `Int' ...
> :t isZero $ intVal 5
isZero $ intVal 5 :: Expr e => e Bool
```

GADTless evaluator 2/3

Evaluation is implemented using a helper data type:

```
newtype Eval a = Eval {runEval :: a}

instance Expr Eval where
    intVal x   = Eval x
    boolVal x  = Eval x
    add x y    = Eval $ runEval x + runEval y
    isZero x   = Eval $ runEval x == 0
    if' x y z = if (runEval x) then y else z
```

```
> runEval $ isZero $ intVal 5
False
> runEval $ isZero $ intVal 0
True
> :t runEval $ isZero $ intVal 0
runEval $ isZero $ intVal 0 :: Bool
```

GADTless evaluator 3/3

Moreover, we can also easily appropriate the evaluator:

```haskell
newtype Print a = Print {printExpr :: String}

instance Expr Print where
    intVal x  = Print $ show x
    boolVal x = Print $ show x
    add x y   = Print $ printExpr x ++ "+" ++ printExpr y
    isZero x  = Print $ "isZero(" ++ printExpr x ++ ")"
    if' x y z = Print $ "if (" ++ printExpr x ++ ") then (" ++
        printExpr y ++ ") else (" ++ printExpr z ++ ")"
```

```
> printExpr $ isZero $ intVal 5
"isZero(5)"
```

Generic programming with GADT

Binary encoder 1/3

Goal:

- encode data in binary form
- the encoder must be able to work with values of several types

```
data Type t where
    TInt  :: Type Int
    TChar :: Type Char
    TList :: Type t -> Type [t]
```

`Type` is a *representation type* (values represent types)

```
> :t TInt
TInt :: Type Int
> :t TList
TList :: Type t -> Type [t]
> :t TList TInt
TList TInt :: Type [Int]
```

Binary encoder 2/3

`String` type is defined as list of `Char` elements:

```
tString :: Type String
tString = TList TChar
```

The output will be a sequence of bits:

```
data Bit = F | T deriving (Eq, Show)
```

Now we can define the encoder:

```
encode :: Type t -> t -> [Bit]
encode TInt i = encodeInt i
encode TChar c = encodeChar c
encode (TList _) [] = F : []
encode (TList t) (x : xs) = T : (encode t x) ++ encode (TList t) xs
```

Binary encoder 3/3

```
encode :: Type t -> t -> [Bit]
encode TInt i = encodeInt i
encode TChar c = encodeChar c
encode (TList _) [] = F : []
encode (TList t) (x : xs) = T : (encode t x) ++ encode (TList t) xs
```

Let's test the evaluator:

```
> encode TInt 333
[T,F,T,F,F,T,T,F,T]
> encode (TList TInt) [1,2,3]
[T,T,T,T,F,T,T,T,F]
> encode tString "test"
[T,T,T,T,F,T,F,F,T,T,T,F,F,T,F,T,T,T,T,T,F,F,T,T,T,T,T,T,F,T,F,F,F]
```

Universal data type

If we *pair* the *representation type* and the *value*, we obtain a universal data type `Dynamic`:

```
data Dynamic where
    Dyn :: Type t -> t -> Dynamic
```

Example of use:

```
encode' :: Dynamic -> [Bit]
encode' (Dyn t v) = encode t v
```

```
> encode' $ Dyn (TList TInt) [5,4,3]
[T,T,F,T,T,T,F,F,T,T,T,F]
```

We can also use `Dynamic` to define heterogenous lists:

```
d = [Dyn TInt 10, Dyn TString "test"]
```

`Dynamic` is useful e.g. for communicating with the environment, when the actual type of data is not known in advance.