Universiteit Utrecht

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

Applied Functional Programming

USCS 2009

Andres Löh

Department of Information and Computing Sciences
Utrecht University

August 26, 2009

C9. Generalized Algebraic Datatypes (GADTs)

C9.1 Motivation

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.

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 the type T,
- result in a simple type, i.e., $T\ a_1 \ldots a_n$ where $a_1, \ldots, a_n$ are distinct type variables.

Another example

$$\textbf{data}\ \text{Either} :: * \longrightarrow * \longrightarrow * \ \textbf{where}$$

Another example

```
data Either :: * → * → * where
  Left  :: a → Either a b
  Right :: b → Either a b
```

Another example

```
data Either :: * → * → * where
  Left  :: a → Either a b
  Right :: b → Either a b
```

Both constructors target Either a b.

Another example

```
data Either :: * → * → * where
  Left  :: a → Either a b
  Right :: b → Either a b
```

Both constructors target Either a b.

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

Evaluation

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

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

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?

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

$$
\begin{aligned}
&\text{eval} :: \text{Expr a} \rightarrow \text{a} \\
&\text{eval} \ldots \\
&\text{eval (Pair x y)} = (\text{eval x}, \text{eval y}) \\
&\text{eval (Fst p)} \quad = \text{fst (eval p)} \\
&\text{eval (Snd p)} \quad = \text{snd (eval p)}
\end{aligned}
$$

C9.2 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 $(+\!\!+)$?

C9.3 Problematic functions

Problematic functions

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

$$(+\!\!+) :: \text{Vec a m} \rightarrow \text{Vec a n} \rightarrow \text{Vec a (Sum m n)}$$

Do we need functions on the type level?

Converting from lists to vectors:

$$\text{fromList} :: [\text{a}] \rightarrow \text{Vec a n}$$

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
```

C9.4 The Equality type

Comparing the length of vectors

```
equalLength :: Vec a m → Vec b n → Bool
```

Comparing the length of vectors

```
equalLength :: Vec a m → Vec b n → Bool
```

Not useful, because

```
if equalLength v w then head (zipWith (,) v w)
                   else ...
```

will not type check. We lose the information that m and n are equal because we return only a Bool!

Equality type

```
data Equal :: * → * → * where
  Refl :: Equal a a
```

Equality type

```
data Equal :: * → * → * where
  Refl :: Equal a a

equalLength :: Vec a m → Vec b n → Maybe (Equal m n)
equalLength Nil              Nil      = Just Refl
equalLength (Cons x xs) (Cons y ys) =
  case equalLength xs ys of
    Just Refl                         → Just Refl
    Nothing                           → Nothing
equalLength _                _        = Nothing
```

Equality type

```
data Equal :: * → * → * where
  Refl :: Equal a a

equalLength :: Vec a m → Vec b n → Maybe (Equal m n)
equalLength Nil              Nil      = Just Refl
equalLength (Cons x xs) (Cons y ys) =
  case equalLength xs ys of
    Just Refl                         → Just Refl
    Nothing                           → Nothing
equalLength _                _        = Nothing

test :: Vec a m → Vec b (Succ n) → (a, b)
test v w = case equalLength v w of
             Just Refl    → head (zipWith (,) v w)
```

Summary

- GADTs can be used to encode advanced properties of types in the type language.
- We end up mirroring expression-level concepts on the type level (e.g. natural numbers).
- GADTs can also represent data that is computationally irrelevant and just guides the type checker (equality proofs, evidence for addition).
  Such information could ideally be erased, but in Haskell, we can always cheat via $\perp$:

  > $\perp$ :: Equal Int Bool