Utrecht University

Advanced Functional Programming

07 - GADTs

Wouter Swierstra & Trevor L. McDonell

Utrecht University

Generalized algebraic data types (GADTs)

A datatype

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

This definition introduces:

A datatype

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

This definition introduces:

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

A datatype

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

This definition introduces:

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

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

A datatype

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

This definition introduces:

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

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

- the possibility 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 a where
  Leaf :: Tree a
  Node :: Tree a -> a -> Tree a -> Tree a
```

Question

What are the *restrictions* regarding the types of the constructors?

Algebraic datatypes

In general, constructor types of an algebraic datatype T:

1. have a return type that is fully applied (of kind $*$):

   ```
   T a1 ... an
   ```

Algebraic datatypes

In general, constructor types of an algebraic datatype T:

1. have a return type that is fully applied (of kind `*`):

   ```
   T a1 ... an
   ```

2. such that `a1, ..., an` are distinct type variables

Algebraic datatypes

In general, constructor types of an algebraic datatype T:

1. have a return type that is fully applied (of kind `*`):

   ```
   T a1 ... an
   ```

2. such that `a1, ..., an` are distinct type variables

3. only use type variables that appear in the return type

Algebraic datatypes

In general, constructor types of an algebraic datatype T:

1. have a return type that is fully applied (of kind `*`):

   `T a1 ... an`

2. such that `a1, ..., an` are distinct type variables

3. only use type variables that appear in the return type

Example:

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

Lifting restrictions

1. have a return type that is fully applied (of kind ∗):

   `T a1 ... an`

2. **such that a1, ..., an are distinct type variables**
3. **only use type variables that appear in the return type**

Does it make sense to lift these restrictions?

Excursion: Expression language

Excursion: Expression language

Imagine we're implementing a small programming language in Haskell:

```haskell
data Expr = LitI   Int
          | LitB   Bool
          | IsZero Expr
          | Plus   Expr Expr
          | If     Expr Expr Expr
```

Excursion: Expression language

Equivalently, we could define the data type as follows:

```
data Expr where
  LitI   :: Int  -> Expr
  LitB   :: Bool -> Expr
  IsZero :: Expr -> Expr
  Plus   :: Expr -> Expr -> Expr
  If     :: Expr -> Expr -> Expr -> Expr
```

Syntax: concrete vs abstract

Imagined concrete syntax:

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

Abstract syntax:

```
If (IsZero (Plus (LitI 0) (LitI 1)))
   (LitB False)
   (LitB True)
```

Type errors

It is all too easy to write ill-typed expressions such as:

```
If (LitI 0) (LitB False) (LitI 1)
```

How can we prevent programmers from writing such terms?

Phantom types

At the moment, *all* expressions have the same type:

```
data Expr = LitI Int
          | LitB Bool
          | ...
```

We would like to distinguish between expressions of *different* types.

Phantom types

At the moment, *all* expressions have the same type:

```
data Expr = LitI Int
          | LitB Bool
          | ...
```

We would like to distinguish between expressions of *different* types.

To do so, we add an additional *type parameter* to our expression data type.

Phantom types

```haskell
data Expr a = LitI   Int
            | LitB   Bool
            | IsZero (Expr Int)
            | Plus   (Expr Int) (Expr Int)
            | If     (Expr Bool) (Expr a) (Expr a)
```

Note that the type variable a is never actually used in the data type for expressions.

We call such type variables *phantom types*.

Constructing well typed terms

Rather than expose the constructors of our expression language, we can instead provide a *well-typed API* for users to write terms:

```
litI :: Int -> Expr Int
litI = LitI

plus :: Expr Int -> Expr Int -> Expr Int
plus = Plus

isZero :: Expr Int -> Expr Bool
isZero = IsZero
```

This guarantees that users will only ever construct well-typed terms!

But, what about writing an interpreter for these expressions?

Evaluation

Before we write an interpreter, we need to choose the type that it returns.

Our expressions may evaluate to booleans or integers:

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

Defining an interpreter now boils down to defining a function:

```
eval :: Expr a -> Val
```

Evaluation

```haskell
eval :: Expr a -> Val
eval (LitI n)   = VInt n
eval (LitB 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"
```

Evaluation (contd.)

- Evaluation code is mixed with code for handling type errors.
- The evaluator uses *tags* (i.e., constructors `VInt`, `VBool`) to distinguish values—these tags are maintained and checked *at runtime*.
- Type errors can, of course, be prevented by writing a type checker for our embedded language, or using phantom types.
- Even if we know that we only have type-correct terms, the Haskell compiler does not enforce this.

Beyond phantom types

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

```
data Expr a where
  LitI   :: Int  -> Expr Int
  LitB   :: Bool -> Expr Bool
  IsZero :: Expr Int -> Expr Bool
  Plus   :: Expr Int -> Expr Int -> Expr Int
  If     :: Expr Bool -> Expr a -> Expr a -> Expr a
```

Each expression has an additional *type argument*, representing the type it will evaluate to.

GADTs

GADTs lift the restriction that all constructors must produce a value of the same type.

- Constructors may have more specific return types
- Pattern matching causes *type refinement*
- Interesting consequences for pattern matching:
  - when case-analyzing an `Expr Int`, it could not be constructed by `LitB` or `IsZero`;
  - when case-analyzing an `Expr Bool`, it could not be constructed by `LitI` or `Plus`;
  - when case-analyzing an `Expr a`, once we encounter the constructor `IsZero` in a pattern, we know that we must be dealing with an `Expr Bool`;
  - …

Evaluation revisited

```haskell
eval :: Expr a -> a
eval (LitI n)      = n
eval (LitB b)      = b
eval (IsZero e)    = eval e == 0
eval (Plus e1 e2) = eval e1 + eval e2
eval (If e1 e2 e3)
  | eval e1        = eval e2
  | otherwise      = eval e3
```

- No possibility for run-time failure; no *tags* required for the return value
- Pattern matching on a GADT requires a type signature. Why?

Limitation: type signatures are required

```
data X a where
  C :: Int -> X Int
  D :: X a
  E :: Bool -> X Bool

f (C n) = [n]            -- (1)
f D     = []             -- (2)
f (E n) = [n]            -- (3)
```

Limitation: type signatures are required

```haskell
data X a where
  C :: Int -> X Int
  D :: X a
  E :: Bool -> X Bool

f (C n) = [n]           -- (1)
f D     = []            -- (2)
f (E n) = [n]           -- (3)
```

What is the type of `f`, with/without (3)? What is the (probable) desired type?

```haskell
f :: X a -> [Int]       -- (1) only
f :: X b -> [c]         -- (2) only
f :: X a -> [Int]       -- (1) + (2)
```

Extending our language

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

```
data Expr a where
    ...
    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’—that is, it is not visible from the type of the compound expression.

Complete the definition

```haskell
data Expr a where
    ...
    Pair :: Expr a -> Expr b -> Expr (a,b)
    Fst  :: Expr (a,b)       -> Expr a
    Snd  :: Expr (a,b)       -> Expr b

eval :: Expr a -> a
eval ...
eval (Pair x y) =
eval (Fst p)    =
eval (Snd p)    =
```

Evaluation again

```haskell
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)
```

GADTs

GADTs have become one of the more popular Haskell extensions.

The classic example for motivating GADTs is the type-safe interpreter, such as the one we have seen here.

However, these richer data types offer many other applications.

In particular, they let us *program* with types in interesting new ways.

Lists with known length

Prelude.head: empty list

```
> myComplicatedFunction 42 "inputFile.csv"
*** Exception: Prelude.head: empty list
```

Can we use the *type system* to rule out such exceptions before a program is run?

Prelude.head: empty list

```
> myComplicatedFunction 42 "inputFile.csv"
*** Exception: Prelude.head: empty list
```

Can we use the *type system* to rule out such exceptions before a program is run?

To do so, we'll introduce a new list-like datatype that records the *length* of the list in its *type*.

Natural numbers and vectors

Natural numbers can be encoded as types (no constructors are required):

```
data Zero
data Succ n
```

Define a vector as a list with a fixed number of elements:

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

Type-safe head and tail

```haskell
head :: Vec a (Succ n) -> a
head (Cons x xs) = x

tail :: Vec a (Succ n) -> Vec a n
tail (Cons x xs) = xs
```

Question

Why is there no case for `Nil` is required?

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
```

Question

Why is there 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 f Nil         Nil         =  Nil
zipWith f (Cons x xs) (Cons y ys) =
  Cons (f x y) (zipWith f xs ys)
```

We can require that the two vectors have the same length!

This lets us rule out bogus cases.

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 (reverse xs) x
```

What about appending two vectors, analogous to the (++) operation on lists?

Problematic functions

- What is the type of our append function?

```
vappend :: Vec a m -> Vec a n -> Vec a ???
```

Problematic functions

- What is the type of our append function?

```
vappend :: Vec a m -> Vec a n -> Vec a ???
```

How can we add two *types*, `n` and `m`?

Problematic functions

- What is the type of our append function?

```
vappend :: Vec a m -> Vec a n -> Vec a ???
```

How can we add two *types*, `n` and `m`?

- Suppose we want to convert from lists to vectors:

```
fromList :: [a] -> Vec a n
```

Where does the type variable `n` come from? What possible values can it have?

Writing vector append

There are multiple options to solve that problem:

- construct explicit evidence; or
- use a type family (more on that in the next lecture).

Explicit evidence

Given two 'types' `m` and `n`, what is their sum?

We can define a GADT describing the *graph* of addition:

```
data Sum m n s where
  SumZero ::                  Sum Zero     n n
  SumSucc :: Sum m n s -> Sum (Succ m) n (Succ s)
```

Explicit evidence

Given two 'types' `m` and `n`, what is their sum?

We can define a GADT describing the *graph* of addition:

```
data Sum m n s where
  SumZero ::                Sum Zero     n n
  SumSucc :: Sum m n s -> Sum (Succ m) n (Succ s)
```

Using this function, we can now define append as follows:

```
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)
```

Passing explicit evidence

This approach has one major disadvantage: we must construct the evidence—the values of type `Sum m n p`—by hand every time we wish to call `append`.

Sometimes we can use fancy type class machinery to automate this construction.

Converting between lists and vectors

It is easy enough to convert from a vector to a list:

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

This simply discards the type information we have carefully constructed.

Converting between lists and vectors

Converting in the other direction, however is not as easy:

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

Question

This definition will not type check. Why?

Converting between lists and vectors

Converting in the other direction, however is not as easy:

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

Question

This definition will not type check. Why?

The type says that the result must be polymorphic in n, that is, it returns a vector of *any* length, rather than a vector of a specific (unknown) length.

From lists to vectors

We can

- specify the length of the vector being constructed in a separate argument; or
- hide the length using an *existential* type.

From lists to vectors (contd.)

Suppose we simply pass in a regular natural number, `Nat`:

```
data Nat = Zero | Succ Nat
```

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

From lists to vectors (contd.)

Suppose we simply pass in a regular natural number, `Nat`:

```
data Nat = Zero | Succ Nat

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

This still does not solve our problem – there is no connection between the natural number that we are passing and the `n` in the return type.

Singletons

We need to reflect type-level natural numbers on the value level.

To do so, we define yet another variation on natural numbers:

```
data Zero
data Succ n

data SNat n where
  SZero ::            SNat Zero
  SSucc :: SNat n -> SNat (Succ n)
```

This is a *singleton type*—for any n, the type `SNat n` has a single inhabitant (the number n).

From lists to vectors

```
data SNat n where
  SZero ::           SNat Zero
  SSucc :: SNat n -> SNat (Succ n)

fromList :: SNat n -> [a] -> Vec a n
fromList SZero     []     = Nil
fromList (SSucc n) (x:xs) = Cons x (fromList n xs)
fromList _         _      = error "wrong length!"
```

Question

This function may still fail dynamically. Why?

From lists to vectors

We can

- specify the length of the vector being constructed in a separate argument; or
- hide the length using an *existential* type.

What about the second alternative?

From lists to vectors

We can define a wrapper around vectors, *hiding* their length:

```
data VecAnyLen a where
  VecAnyLen :: Vec a n -> VecAnyLen a
```

A value of type `VecAnyLen a` stores a vector of *some* length with values of type `a`.

From lists to vectors

We can convert any list to a vector of some length as follows:

```
fromList :: [a] -> VecAnyLen a
fromList []     = VecAnyLen Nil
fromList (x:xs) =
  case fromList xs of
    VecAnyLen ys -> VecAnyLen (Cons x ys)
```

From lists to vectors

We can combine the two approaches and include a `SNat` in the packed type:

```
data VecAnyLen a where
  VecAnyLen :: SNat n -> Vec a n -> VecAnyLen a
```

Question

How does the conversion function change?

Does this tell us anything new?

Comparing the length of vectors

We can define a boolean function that checks when two vectors have the same length

```
equalLength :: Vec a m -> Vec b n -> Bool
equalLength Nil         Nil         = True
equalLength (Cons _ xs) (Cons _ ys) = equalLength xs ys
equalLength _           _           = False
```

Comparing the length of vectors

Suppose I want to use this to check the lengths of my vectors:

```
if equalLength xs ys
  then zipVec xs ys
  else error "Wrong lengths"
```

```
zipVec :: Vec a n -> Vec b n -> Vec (a,b) n
```

Question

Will this type check?

Comparing the length of vectors

Suppose I want to use this to check the lengths of my vectors:

```
if equalLength xs ys
  then zipVec xs ys
  else error "Wrong lengths"
```

```
zipVec :: Vec a n -> Vec b n -> Vec (a,b) n
```

Question

Will this type check?

No! When `equalLength xs ys` returns True, this does not provide any *type level* information that `m` and `n` are equal.

How can we enforce that two types are indeed equal?

Equality type

Equality type

Just as we saw for the `Sum` type, we can introduce a GADT that witnesses that two types are equal:

```
data Equal a b where
  Refl :: Equal a a
```

Pattern matching on `Refl` produces a proof that `a ~ b`.

Properties of the equality relation

`Equal` is an equivalence relation.

```
refl  :: Equal a a
sym   :: Equal a b -> Equal b a
trans :: Equal a b -> Equal b c -> Equal a c
```

How are these functions defined?

Properties of the equality relation

`Equal` is an equivalence relation.

```
refl  :: Equal a a
sym   :: Equal a b -> Equal b a
trans :: Equal a b -> Equal b c -> Equal a c
```

How are these functions defined?

```
refl            = Refl
sym   Refl      = Refl
trans Refl Refl = Refl
```

Build an equality proof

Instead of returning a boolean, we can now provide evidence that the length of two vectors is equal:

```
eqLength :: Vec a m -> Vec b n -> Maybe (Equal m n)
eqLength Nil          Nil          = Just Refl
eqLength (Cons x xs) (Cons y ys)
  | Just Refl <- eqLength xs ys
  = Just Refl
eqLength _            _            = Nothing
```

Using equality

```
test :: Vec a m -> Vec b (Succ n) -> Maybe (a,b)
test xs ys =
  case eqLength xs ys of
    Just Refl -> Just (head (zipVec xs ys))
    _         -> Nothing
```

Question

Why does this type check?

Expressive power of equality

The equality type can be used to encode other GADTs.

Recall our expression example using phantom types:

```
data Expr a = LitI   Int
            | LitB   Bool
            | IsZero (Expr Int)
            | Plus   (Expr Int) (Expr Int)
            | If     (Expr Bool) (Expr a) (Expr a)
```

Expressive power of equality

We can use equality proofs and phantom types to implement (some) GADTs:

```
data Expr a
  = LitI   (Equal a Int)  Int
  | LitB   (Equal a Bool) Bool
  | IsZero (Equal a Bool) (Expr Int)
  | Plus   (Equal a Int)  (Expr Int) (Expr Int)
  | If                    (Expr Bool) (Expr a) (Expr a)
```

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, but is used to guide the type checker (equality proofs, evidence for addition).