COMP3141

Software System Design and Implementation

GADTs Practice

Christine Rizkallah
CSE, UNSW (and Data61)
Term 2 2019

Recall: GADTs

Generalised Algebraic Datatypes (*GADTs*) is an extension to Haskell that, among other things, allows data types to be specified by writing the types of their constructors:

```haskell
{-# LANGUAGE GADTs, KindSignatures #-}
-- Unary natural numbers, e.g. 3 is S (S (S Z))
data Nat = Z | S Nat
-- is the same as
data Nat :: * where
  Z :: Nat
  S :: Nat -> Nat
```

The printf function

Consider the well known C function printf:

The printf function

Consider the well known C function printf:

```
printf("Hello %s You are %d years old!", "Nina", 22)
```

The `printf` function

Consider the well known C function `printf`:

```
printf("Hello %s You are %d years old!", "Nina", 22)
```

In C, the type (and number) of parameters passed to this function are dependent on the first parameter (the format string).

The `printf` function

To do a similar thing in Haskell, we would like to use a richer type that allows us to define a function whose subsequent parameter is determined by the first.

The printf function

To do a similar thing in Haskell, we would like to use a richer type that allows us to define a function whose subsequent parameter is determined by the first.

```
data Format :: * -> * where
  End :: Format ()                              -- Empty format
  Str :: Format t -> Format (String, t)  -- %s
  Dec :: Format t -> Format (Int, t)      -- %d
  L :: String -> Format t -> Format t  -- literal strings
  deriving instance Show (Format ts)
  -- just like deriving (Show) for normal data types.
```

Our format strings is indexed by a tuple type containing all of the types of the %directives used.

The `printf` function

```
"Hello %s You are %d years old!"
```

is written:

```
L "Hello" $ Str $ L " You are "
     $ Dec $ L " years old!" End
```

The printf function

```
printf :: Format ts -> ts -> IO ()
printf End () =
    pure () -- type is ()
printf (Str fmt) (s,ts) =
    do putStr s; printf fmt ts -- type is (String, ...)
printf (Dec fmt) (i,ts) =
    do putStr (show i); printf fmt ts -- type is (Int,..)
printf (L s fmt) ts      =
    do putStr s; printf fmt ts
```

Vectors

Define a natural number kind to use on the type level:

```
data Nat = Z | S Nat
```

Vectors

Define a natural number kind to use on the type level:

```
data Nat = Z | S Nat
```

Our length-indexed list can be defined, called a `Vec`:

```
data Vec (a :: *) :: Nat -> * where
  Nil  :: Vec a Z
  Cons :: a -> Vec a n -> Vec a (S n)
```

Vectors

Define a natural number kind to use on the type level:

```
data Nat = Z | S Nat
```

Our length-indexed list can be defined, called a `Vec`:

```
data Vec (a :: *) :: Nat -> * where
  Nil  :: Vec a Z
  Cons :: a -> Vec a n -> Vec a (S n)
```

The functions `hd` and `tl` can be total:

```
hd :: Vec a (S n) -> a
hd (Cons x xs) = x
tl :: Vec a (S n) -> Vec a n
tl (Cons x xs) = xs
```

Vectors, continued

Our `map` for vectors is as follows:

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

Vectors, continued

Our map for vectors is as follows:

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

Properties

Using this type, it's impossible to write a mapVec function that changes the length of the vector.

Properties are verified by the compiler!

Appending Vectors

Example (Problem)

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

Appending Vectors

Example (Problem)

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

We want to write `m + n` in the `???` above, but we do not have addition defined for kind `Nat`.

Appending Vectors

Example (Problem)

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

We want to write `m + n` in the `???` above, but we do not have addition defined for kind `Nat`.

We can define a normal Haskell function easily enough:

```
plus :: Nat -> Nat -> Nat
plus Z y = y
plus (S x) y = S (plus x y)
```

Appending Vectors

Example (Problem)

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

We want to write `m + n` in the `???` above, but we do not have addition defined for kind `Nat`.

We can define a normal Haskell function easily enough:

```
plus :: Nat -> Nat -> Nat
plus Z y = y
plus (S x) y = S (plus x y)
```

This function is not applicable to type-level `Nat`s, though.

Appending Vectors

Example (Problem)

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

We want to write `m + n` in the `???` above, but we do not have addition defined for kind `Nat`.

We can define a normal Haskell function easily enough:

```
plus :: Nat -> Nat -> Nat
plus Z y = y
plus (S x) y = S (plus x y)
```

This function is not applicable to type-level `Nat`s, though.

$\Rightarrow$ we need a type level function.

Type Families

Type level functions, also called *type families*, are defined in Haskell like so:

```
{-# LANGUAGE TypeFamilies #-}
type family Plus (x :: Nat) (y :: Nat) :: Nat where
  Plus Z     y = y
  Plus (S x) y = S (Plus x y)
```

We can use our type family to define appendV:

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

Concatenating Vectors

Example (Problem)

```
concatV :: Vec (Vec a m) n -> Vec a ???
```

Concatenating Vectors

Example (Problem)

```
concatV :: Vec (Vec a m) n -> Vec a ???
```

We want to write `m * n` in the `???` above, but we do not have times defined for kind `Nat`.

Concatenating Vectors

Example (Problem)

```
concatV :: Vec (Vec a m) n -> Vec a ???
```

We want to write `m * n` in the `???` above, but we do not have times defined for kind `Nat`.

Concatenating Vectors

Example (Problem)

```
concatV :: Vec (Vec a m) n -> Vec a ???
```

We want to write `m * n` in the `???` above, but we do not have times defined for kind `Nat`.

```
{-# LANGUAGE TypeFamilies #-}
type family Times (a :: Nat) (b :: Nat) :: Nat where
  Times Z n = Z
  Times (S m) n = Plus n (Times m n)
```

We can use our type family to define `concatV`:

```
concatV :: Vec (Vec a m) n -> Vec a (Times n m)
concatV Nil = Nil
concatV (Cons v vs) = v `appendV` concatV vs
```

Filtering Vectors

Example (Problem)

```
filterV :: (a -> Bool) -> Vec a n -> Vec a ???
```

Filtering Vectors

Example (Problem)

```
filterV :: (a -> Bool) -> Vec a n -> Vec a ???
```

What is the size of the result of filter?

Filtering Vectors

Example (Problem)

```
filterV :: (a -> Bool) -> Vec a n -> [a]
```

We do not know the size of the result.

Filtering Vectors

Example (Problem)

```
filterV :: (a -> Bool) -> Vec a n -> [a]
```

We do not know the size of the result.

We can use our type family to define concatV:

```
filterV :: (a -> Bool) -> Vec a n -> [a]
filterV p Nil = []
filterV p (Cons x xs)
                  | p x   = x : filterV p xs
                  | otherwise = filterV p xs
```

Homework

1. Assignment 2 is **released**. Due on **7th August, 9 AM**.
2. The last programming exercise has been released, due next week.
3. This week's quiz is also up, due in Friday of Week 9.