COMP3141

Software System Design and Implementation

Lecture 9: Generalised Algebraic Data Types

Johannes Åman Pohjola
University of New South Wales
Term 2 2023

Exam Information

Time: 8am-12pm AEST on Monday Aug 14.

Length: 2 hours. **Make sure you start before 10AM**.

Where: online. Link will appear on the course website.

Exam Information

Material: all material that was presented in the course, including in lectures, practicals, exercise sets or quizzes (except where we explicitly told you that the material was not examinable).

Format: There will be quiz-style questions about design. There will be theory questions. We may ask you to write code and proofs, but no long-form software implementation.

Sample exam will be released on the course website shortly.

GADTs

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

```
data Answer = Yes | No
-- is the same as
data Answer :: * where
  Yes :: Answer
  No :: Answer
```

GADTS

We will need to use two new language extensions to declare them.

```
{-# LANGUAGE KindSignatures,
             GADTs,
             StandaloneDeriving #-}
data Parity :: * where -- GADTs
  Even :: Parity
  Odd :: Parity
-- StandaloneDeriving
deriving instance Show Parity
deriving instance Eq Parity
```

Aside: Sum Types

```
data Parity = Even | Odd
data Polarity = Positive | Zero | Negative
data Sum :: * -> * -> * where
  L :: a -> Sum a b
  R :: b -> Sum a b
```

Aside: Sum Types

```
data Parity = Even | Odd
data Polarity = Positive | Zero | Negative
data Sum :: * -> * -> * where
  L :: a -> Sum a b
  R :: b -> Sum a b
```

Questions

1. How many elements does the type `Polarity` have?

Aside: Sum Types

```
data Parity = Even | Odd
data Polarity = Positive | Zero | Negative
data Sum :: * -> * -> * where
  L :: a -> Sum a b
  R :: b -> Sum a b
```

Questions

1. How many elements does the type `Polarity` have?
2. How many elements does `Sum Parity Polarity` have?

Aside: Sum Types

```
data Parity = Even | Odd
data Polarity = Positive | Zero | Negative
data Sum :: * -> * -> * where
  L :: a -> Sum a b
  R :: b -> Sum a b
```

Questions

1. How many elements does the type `Polarity` have?
2. How many elements does `Sum Parity Polarity` have?
3. How many elements does `Sum Polarity Polarity` have?

Aside: Sum Types

```
data Parity = Even | Odd
data Polarity = Positive | Zero | Negative
data Sum :: * -> * -> * where
  L :: a -> Sum a b
  R :: b -> Sum a b
```

Questions

1. How many elements does the type `Polarity` have?
2. How many elements does `Sum Parity Polarity` have?
3. How many elements does `Sum Polarity Polarity` have?

Do we see why they are called sum types?

Aside: Product Types

```
data Parity = Even | Odd
data Polarity = Positive | Zero | Negative
data Prod :: * -> * -> * where
  Pair :: a -> b -> Prod a b
```

Aside: Product Types

```
data Parity = Even | Odd
data Polarity = Positive | Zero | Negative
data Prod :: * -> * -> * where
  Pair :: a -> b -> Prod a b
```

Questions

1. How many elements does the type `Polarity` have?

Aside: Product Types

```
data Parity = Even | Odd
data Polarity = Positive | Zero | Negative
data Prod :: * -> * -> * where
  Pair :: a -> b -> Prod a b
```

Questions

1. How many elements does the type `Polarity` have?
2. How many elements does `Prod Parity Polarity` have?

Aside: Product Types

```
data Parity = Even | Odd
data Polarity = Positive | Zero | Negative
data Prod :: * -> * -> * where
  Pair :: a -> b -> Prod a b
```

Questions

1. How many elements does the type `Polarity` have?
2. How many elements does `Prod Parity Polarity` have?
3. How many elements does `Prod Polarity Polarity` have?

Aside: Product Types

```haskell
data Parity = Even | Odd
data Polarity = Positive | Zero | Negative
data Prod :: * -> * -> * where
  Pair :: a -> b -> Prod a b
```

Questions

1. How many elements does the type `Polarity` have?
2. How many elements does `Prod Parity Polarity` have?
3. How many elements does `Prod Polarity Polarity` have?

Do we see why they are called product types?
NB: here we count *non-bottom* elements; e.g. `undefined` doesn't count.

Sized lists

We can use GADTs+phantom types to encode the length of a list in its type:

```
data Size = Z | S Size

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

Sized lists

We can use GADTs+phantom types to encode the length of a list in its type:

```
data Size = Z | S Size

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

- `Nil` always has length 0 (`Z`)
- `Cons x xs` is one longer than `xs` (`S n`)

Sized lists

We can use GADTs+phantom types to encode the length of a list in its type:

```
data Size = Z | S Size

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

- `Nil` always has length 0 (`Z`)
- `Cons x xs` is one longer than `xs` (`S n`)

Observation

This subsumes the distinct types for empty and non-empty lists we've seen previously.

Sized lists

Look at the type of the map function for `Vec`:

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

It says that if the input has length `n`, then so does the output. So the property that `mapV` preserves length is enforced by the type system!

Think about all the inductive proofs we don't have to write.

Tradeoffs

GADTs are one of the most powerful static assurance tools available in Haskell. But:

- It can be difficult to convince the Haskell type checker that your code is correct, even when it is.
- Type-level encodings can make types more verbose and programs harder to understand.
- Too detailed types can make type-checking very slow, hindering xsproductivity.

Be pragmatic!

Use type-based encodings when the assurance advantages outweigh the potential disadvantages. The typical use case is to eliminate partial functions from our code base

That's all folks!

Thanks for taking the course.

Don't forget to take the myExperience survey.