COMP3141

Software System Design and Implementation

Lecture 9: Generalized Algebraic Data Types

Zoltan A. Kocsis
University of New South Wales
Term 2 2022

Marking

- How will the exercises and quizzes be marked?
- Due to the way the course works, we cannot offer extensions on exercises and quizzes.
- To make up for this, your quiz and exercise marks are determined as follows:
  - **Quiz:** Best 6 out of the 7 quizzes. Your lowest-scoring quiz is not counted.
  - **Exercise Sets:** Best 5 out of 6 exercise sets. Your lowest-scoring exercise set is not counted.

E.g. if you scored $8/8$ in the first six quizzes, and $0/8$ in the last one, you will still get full marks for the quizzes.

Exam Information I

Date: Tuesday, 23 Aug 2022
Where? Online.

How long?

You can start it any time between 0000-2100 (Sydney time). But once you start it, you have **3 hours** to finish.

Exam Information II

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 will ask you to write code and proofs, but no long-form software implementation.

Sample Exam: Will be released on the course website by Monday.

Revision

Raphael has kindly agreed to do some revision with you tomorrow.
Please read and use the megathread on the forums:
`https://edstem.org/au/courses/8705/discussion/951009`

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
```

When combined with phantom types, this becomes a very powerful tool of static assurance!

Simple ADTs as GADTs

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

```
{-# LANGUAGE KindSignatures, GADTs, StandaloneDeriving #-}
```

We need the latter because `deriving (Show, Eq)` etc. does not work with the GADT syntax.

```
data Parity :: * where -- GADTs
  Even :: Parity
  Odd  :: Parity

-- StandaloneDeriving
deriving instance Show Parity
deriving instance Eq Parity
```

Demo: Simple ADTs

Maybe as GADT

Familiar types can be written as GADTs:

```haskell
-- data Maybe a = Nothing | Just a
data Maybe :: * -> * where -- GADTs
  Nothing :: Maybe a
  Just    :: a -> Maybe a
deriving instance (Show a) => Show (Maybe a)
deriving instance (Eq a) => Eq (Maybe a)
```

Notice that deriving is a bit more complicated.

Demo: Declaring Sum/Either

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
```

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: do not count bottom (`undefined, error ""`, infinite loops) elements.

Demo: Product Type

Sized Lists (Vectors)

So far, just a new syntax. But when combined with phantom types and kind signatures, it will enable us to statically assure many properties. Previously we had the type `[a]` of possibly lists, and a separate type for `NonEmptyList a`. Now let's unite them by declaring a list type which knows its size at compile time.

```
data Size = Z | S Size
-- Z represents 0
-- S Z represents 1
-- S (S Z) represents 2, etc.
```

Sized Lists

```
data Size = Z | S Size

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

What does this mean?

- Something constructed using `Nil` has length zero.
- `Cons x xs` is one longer than the argument `xs`.

Observation

Previously, we had to use multiple types to distinguish empty, non-empty and possibly empty lists. There is now only *one* set of precisely-typed constructors that can do the same job.

GADT pattern matching: GHC knows that if we matched against `Nil`, the size must have been zero.

Static Assurance with Size

Let's write a `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)
```

Notice the type signature! It says that if the input has length `n`, then the output has the same length `n`. So `mapV` preserves length!

Demo: mapV, zipV

Static Assurance with Size

Previously: had to prove `length (map f xs) == length xs` manually using induction.

Now: The Haskell type checker ensures (proves) this for us.

```
mapV :: (a -> b) -> Vec a n -> Vec b n
```

Properties

Using this type, it's impossible to write a `mapV` function that changes the length of the vector.
Properties are verified by the compiler!

RPN Calculator, reprise

Exercise 5: RPN calculator with zero padding (no possible error states).

Practical 9: RPN calculator with dynamic error handling (`Staybe` monad).

Now: RPN calculator with static assurance (no possible error states).

Demo: RPN Calculator

Demo: Tying it together (eqn. reasoning)

Tradeoffs

GADTs are one of the most powerful static assurance tools available in Haskell. The benefits of the extra static checking are obvious. However:

- 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.
- Sometimes excessively detailed types can make type-checking very slow, hindering productivity.

Be pragmatic!

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

FIN

1. **A massive thanks to you all!**
2. We hope it wasn't too demanding so far: it's less material than in previous years.
3. Let us know what you liked, what you didn't like by filling in the myExperience survey.

See you soon!