COMP3141

Software System Design and Implementation

Lecture 2: Induction, Data Types, Type Classes

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

Announcements

Quiz 01: The submission issues have been resolved. Submit your solutions on the course website.

Due: Saturday, June 11, 11:59:59 PM.

Reminder: Sets, subsets

The set of natural numbers

$$\mathbb{N} = \{0, 1, 2, 3, 4, 5, \dots\}$$

- The usual counting numbers.
- In this course, they start from zero.

Reminder: Sets, subsets

Set comprehension notation

Consider any mathematical property φ that a natural number may have (e.g. being even, being a prime, being a square number, etc.)
As a first approximation, you can think of a property as a Haskell function

```
phi :: NaturalNumber -> Bool
```

although we won't require properties to be decidable or computable.
The set denoted $\{x \in \mathbb{N} \mid \varphi(x)\}$ consists of all elements x of $\mathbb{N}$ that satisfy the property φ.

Reminder: Sets, subsets

Set comprehension examples

The set denoted $\{x \in \mathbb{N} \mid \varphi(x)\}$ consists of all elements $x \in \mathbb{N}$ that satisfy the property φ.

- $\{x \in \mathbb{N} \mid x \text{ is odd}\} = \{1, 3, 5, 7, 9, \ldots\}$
- $\{x \in \mathbb{N} \mid x \text{ is prime}\} = \{2, 3, 5, 7, 11, \ldots\}$
- $\{x \in \mathbb{N} \mid x < 5\} = \{0, 1, 2, 3, 4\}$
- $\{x \in \mathbb{N} \mid x \leq 4 \text{ and } x \text{ is prime}\} = \{2, 3\}$
- $\{x \in \mathbb{N} \mid x \geq x\} = \{0, 1, 2, 3, 4, 5, \ldots\} = \mathbb{N}$
- $\{x \in \mathbb{N} \mid x \geq x^2\} = \{0, 1\}$
- $\{x \in \mathbb{N} \mid x \geq x + 1\} = \emptyset$.

The Principle of Induction: Sets

Consider any set S of numbers. If

1. the set S contains 0, i.e. $0 \in S$, and
2. whenever S contains some number x, it also contains $x + 1$,

then S contains every natural number.

- By the first property, $0 \in S$.
- By the second property, if $0 \in S$ then $1 \in S$.
- Thus, $1 \in S$.
- By the second property, if $1 \in S$ then $2 \in S$.
- Thus, $2 \in S$.
- and so on...

The Principle of Induction: Properties

Consider any set S of numbers. If

1. the set S contains 0, i.e. $0 \in S$, and
2. whenever S contains some number x, it also contains $x + 1$,

then S contains every natural number.

Consider a property φ and the set $S = \{x \in \mathbb{N} \mid \varphi(x)\}$. Then $y \in S$ precisely if y has the property φ, i.e. if $\varphi(y)$ holds. Rewriting the conditions above, we get that if

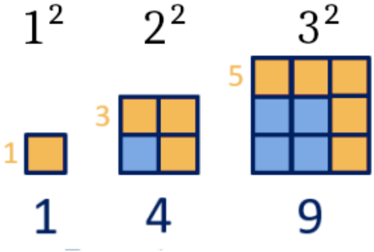
1. $\varphi(0)$ and
2. whenever $\varphi(x)$ holds for some $x \in \mathbb{N}$, so does $\varphi(x + 1)$,

then $\{x \in \mathbb{N} \mid \varphi(x)\}$ contains every natural number, i.e. the property φ holds for every natural number.

Example 1: Induction on $\mathbb{N}$

Problem

Prove that the sum of the first n odd numbers is equal to n^2.



Demo: Proof

Structural Induction

Consider a Haskell type `t` and any set S of lists. If

1. the set S contains `[] ::[t]`, and
2. whenever S contains some list `xs`, then S also contains every list of the form `(x:xs)` (where `x ::t`),

then S contains all[1] lists of type `[t]`.

[1] Haskell is a lazy language: really, we should say all finite lists

Structural Induction

If we want to prove that a property $\varphi(\texttt{xs})$ holds for all lists `xs`, it suffices to prove that:

1. the empty list satisfies that property, i.e. $\varphi([\,])$; and
2. whenever a list `ys` satisfies $\varphi(\texttt{ys})$, so does any list of the form `(x:ys)`.

This is known as **Proof by Structural Induction** on the list structure.

Demo: `map` preserves the length of its input

Properties of Programs

- Reasoning about functional programs:
  equational reasoning + structural induction
- Structural induction: works over lists and other data types
- This course: simple induction proofs over $\mathbb{N}$ and lists.
- For more: **COMP3161**.

Enumerated Data Types

100 pts of ID

When applying for a bank account in NSW, you have to provide documents used to verify your identity. Each document is worth some points, and you need a total of 100 or more points to successfully verify your identity.

Real-life example:

- **Primary documents**: *Passport* or *Birth Certificate*. Each worth 70 pts.
- **Secondary:** *Driver's License* or *Student ID*. The first document used from this list is worth 40 pts, any additional items 25 pts.
- **Tertiary:** Existing *credit cards*. Worth 25 pts.

Enumerated Data Types

Task 1

You work for a bank. Your task is to write a program that calculates the total point value of a given list of documents.

Demo: Enumerated data types

Compound Data Types

While working with days of a month, you might use a type like this:

```haskell
type MonthDay = (Int, Int)  -- (month, day)
```

Notice that:

- Nothing distinguishes your `Int`-pair from any other `Int`-pair.
- You can provide e.g. a pair of image coordinates to a function that expects a `MonthDay`: static type checking does not work for you.

Compound Data Types

Instead, you can use `data`

```
data MonthDay = MonthDay Int Int
```

…or better yet…

```
type Day = Int
data Month = Jan | Feb | Mar | ...
data MonthDay = MonthDay Month Day
```

Demo: MonthDay, showMonthDay

Multiple Constructors

We can of course have multiple constructors. Types with more than one constructor are sometimes called *sum types*. Example: Zoom meetings.

```
data WeekDay = Mon | Tue | Wed | ...
data ZoomMeetingTime
  = Once Year MonthDay
  | RecurringWeekly WeekDay
```

Recursive and Parametric Types

Data types can also be defined with parameters, such as the well known `Maybe` type, defined in the standard library:

```
data Maybe a = Just a | Nothing
```

Types can also be recursive. If lists weren't already defined in the standard library, we could define them ourselves:

```
data List a = Nil | Cons a (List a)
```

We can even define natural numbers, where 2 is encoded as `Succ(Succ Zero)`:

```
data Natural = Zero | Succ Natural
```

Types in Design

Sage Advice

An old adage due to Yaron Minsky (of Jane Street) is:

Make illegal states unrepresentable.

Choose types that *constrain* your implementation as much as possible. Then failure scenarios are eliminated automatically.

Example (Contact Details)

```
data Contact = C Name (Maybe Address) (Maybe Email)
```

is changed to:

```
data ContactDetails = EmailOnly Email
                    | PostOnly Address
                    | Both Address Email
data Contact = C Name ContactDetails
```

What failure state is eliminated here?

Partial Functions

Failure to follow Yaron's excellent advice leads to partial functions.

Definition

A *partial function* is a function not defined for all possible inputs.
Examples: `head`, `tail`, `(!!)`, division

Partial functions are to be avoided, because they cause your program to crash if undefined cases are encountered.
To eliminate partiality, we must either:

- enlarge the codomain, usually with a `Maybe` type:

```
safeHead :: [a] -> Maybe a -- Q: How is this safer?
safeHead (x:xs) = Just x
safeHead []     = Nothing
```

- Or we must constrain the domain to be more specific:

```
safeHead' :: NonEmpty a -> a -- Q: How to define?
```

Demo: defining NonEmpty

Parse, don't validate

```haskell
safeHead :: [a] -> Maybe a
safeHead (x:xs) = Just x
safeHead []     = Nothing
safeHead' :: NonEmpty a -> a
safeHead' (One x _) = x
safeHead' (Cons x _) = x
```

Sage Advice

A slogan from Alexis King:

Parse, don't validate.

Means:

- Validation function should return structured data which cannot represent illegal states (parse).
- Other functions should take only input types they can safely consume (don't validate)

Type Classes

You have already seen functions such as:

- `compare`
- `(==)`
- `(+)`
- `show`

that work on multiple types, and their corresponding constraints on type variables `Ord`, `Eq`, `Num` and `Show`.

These constraints are called *type classes*, and can be thought of as a set of types for which certain operations are implemented.

Show

The `Show` type class is a set of types that can be converted to strings. It is defined like:

```
class Show a where  -- nothing to do with OOP
  show :: a -> String
```

Types are added to the type class as an *instance* like so:

```
instance Show Bool where
  show True  = "True"
  show False = "False"
```

We can also define instances that depend on other instances:

```
instance Show a => Show (Maybe a) where
  show (Just x) = "Just " ++ show x
  show Nothing  = "Nothing"
```

Fortunately for us, Haskell supports automatically `deriving` instances for some classes, including `Show`.

Semigroup

Semigroups

A *semigroup* is a pair of a set S and an operation $\bullet : S \to S \to S$ where the operation $\bullet$ is *associative*.
Associativity is defined as, for all a, b, c:

$$(a \bullet (b \bullet c)) = ((a \bullet b) \bullet c)$$

Haskell has a type class for semigroups! The associativity law is enforced only by programmer discipline:

```
class Semigroup s where
  (<>) :: s -> s -> s
  -- Law: (<>) must be associative.
```

What instances can you think of?

Semigroup

Let's implement additive (RGB) colour mixing:

```
data Color = Color Int Int Int Int
  -- Red, Green, Blue, Alpha (transparency)
instance Semigroup Color where
  (Color r1 g1 b1 a1) <> (Color r2 g2 b2 a2)
      = Color (mix r1 r2)
              (mix g1 g2)
              (mix b1 b2)
              (mix a1 a2)
    where
      mix x1 x2 = min 255 (x1 + x2)
```

Associativity is satisfied.

Monoid

Monoids

A *monoid* is a semigroup $(S, \bullet)$ equipped with a special *identity element* $z : S$ such that $x \bullet z = x$ and $z \bullet y = y$ for all x, y.

```
class (Semigroup a) => Monoid a where
  mempty :: a
```

For colours, the identity element is transparent black:

```
instance Monoid Color where
  mempty = Color 0 0 0 0
```

For each of the semigroups discussed previously:

- Are they monoids?
- If so, what is the identity element?

Are there any semigroups that are not monoids?

Non-empty lists, maximum

Newtypes

There are multiple possible monoid instances for numeric types like `Integer`:

- The operation `(+)` is associative, with identity element `0`
- The operation `(*)` is associative, with identity element `1`

Haskell doesn't use any of these, because there can be only one instance per type per class in the entire program (including all dependencies and libraries used).

A common technique is to define a separate type that is represented identically to the original type, but can have its own, different type class instances.

In Haskell, this is done with the `newtype` keyword.

Newtypes

A `newtype` declaration is much like a `data` declaration except that there can be only one constructor and it must take exactly one argument:

```
newtype Score = S Integer

instance Semigroup Score where
  S x <> S y = S (x + y)

instance Monoid Score where
  mempty = S 0
```

Here, `Score` is represented identically to `Integer`, and thus no performance penalty is incurred to convert between them.

In general, newtypes are a great way to prevent mistakes. Use them frequently!

Ord

`Ord` is a type class for inequality comparison:

```
class Ord a where
  (<=) :: a -> a -> Bool
```

What laws should instances satisfy?

For all `x`, `y`, and `z`:

1. *Reflexivity*: `x <= x`.
2. *Transitivity*: If `x <= y` and `y <= z` then `x <= z`.
3. *Antisymmetry*: If `x <= y` and `y <= x` then `x == y`.
4. *Totality*: Either `x <= y` or `y <= x`

Relations that satisfy these four properties are called *total orders*. Without the fourth (totality), they are called *partial orders*.

Eq

Eq is a type class for equality or equivalence:

```
class Eq a where
  (==) :: a -> a -> Bool
```

What laws should instances satisfy?

For all x, y, and z:

1. *Reflexivity*: x == x.
2. *Transitivity*: If x == y and y == z then x == z.
3. *Symmetry*: If x == y then y == x.

Relations that satisfy these are called *equivalence relations*.

Some argue that the Eq class should be only for *equality*, requiring stricter laws like:

If x == y then f x == f y for all functions f

But this is debated.

FIN

Assigned reading: Alexis King - Parse, don't validate (Blog Post) `https://lexi-lambda.github.io/blog/2019/11/05/parse-don-t-validate/` You don't have to understand all the example code, but you should familiarize yourself with the ideas in the blog post.

1. Don't forget to submit Quiz 1.
2. Exercise 1 and Quiz 2 will be released tomorrow.