COMP3141

Software System Design and Implementation

Lecture 2: Induction, Data Types, Type Classes

Johannes Åman Pohjola
University of New South Wales
Term 2 2023

Announcements

Quiz 01: You still have until tomorrow 11:59:59 PM to do it.

Recap: Induction

Suppose we want to prove that a property $P(n)$ holds for all natural numbers n.
Remember that the set of natural numbers $\mathbb{N}$ can be defined as follows:

Definition of Natural Numbers

1. 0 is a natural number.
2. For any natural number n, $n + 1$ is also a natural number.

Recap: Induction

Therefore, to show $P(n)$ for all n, it suffices to show:

1. $P(0)$ (the *base case*), and
2. assuming $P(k)$ (the *inductive hypothesis*),
   $\Rightarrow P(k+1)$ (the *inductive case*).

Example

Show that $f(n) = n^2$ for all $n \in \mathbb{N}$, where:

$$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ 2n - 1 + f(n-1) & \text{if } n > 0 \end{cases}$$

Induction on Lists

Haskell lists can be defined similarly to natural numbers.

Definition of Haskell Lists

1. `[]` is a list.
2. For any list `xs`, `x:xs` is also a list (for any item `x`).

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

Induction on Lists

Haskell lists can be defined similarly to natural numbers.

Definition of Haskell Lists

1. `[]` is a list.
2. For any list `xs`, `x:xs` is also a list (for any item `x`).

This means, if we want to prove that a property $P(\texttt{ls})$ holds for all lists `ls`[1], it suffices to show:

1. $P(\texttt{[]})$ (the base case)
2. $P(\texttt{x:xs})$ for all items x, assuming the inductive hypothesis $P(\texttt{xs})$.

Demo: `map` preserves the length of its input

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

Properties of Programs

- Reasoning about functional programs:
  equational reasoning + structural induction

Properties of Programs

- Reasoning about functional programs:
  equational reasoning + structural induction
- Structural induction: works over lists and other data types

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**, **COMP4161**.

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.

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...

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
```

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
```

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

Types can have type parameters:

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

Recursive and Parametric Types

Types can have type parameters:

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

Types can be recursive:

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

Recursive and Parametric Types

Types can have type parameters:

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

Types can be recursive:

```
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 by construction.

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 by construction.

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 should be avoided, because they can crash your program. How do we eliminate partiality?

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 should be avoided, because they can crash your program. How do we eliminate partiality?

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

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

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 should be avoided, because they can crash your program. How do we eliminate partiality?

- We can 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 can constrain the domain to be more specific:

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

Parse, don't validate

```
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.

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`.

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
```

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"
```

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*.

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.

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
```

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?

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?

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`

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.

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?

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`.

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`.

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`.

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`

Ord

`Ord` is a type class for inequality comparison:

```haskell
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?

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.

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.

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`.

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*.

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.