Structural Induction with Haskell

Johannes Åman Pohjola
UNSW
Term 3 2022
Recap: Induction

**Definition**

Let $P(x)$ be a predicate on natural numbers $x \in \mathbb{N}$. To show $\forall x \in \mathbb{N}. \ P(x)$, we can use induction:

- Show $P(0)$
- Assuming $P(k)$ (the *inductive hypothesis*), show $P(k + 1)$.

**Example (Sum of Integers)**

Write a recursive function $\text{sumTo}$ to sum up all integers from 0 to the input $n$.

Show that:

$$\forall n \in \mathbb{N}. \ \text{sumTo} \ n = \frac{n(n + 1)}{2}$$
Haskell Data Types

We can define natural numbers as a Haskell data type, reflecting this inductive structure.

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

**Example**

Define addition, prove that $\forall n. n + Z = n$.

**Inductive Structure**

Observe that the non-recursive constructors correspond to base cases and the recursive constructors correspond to inductive cases.
Lists

Lists are *singly-linked* lists in Haskell. The empty list is written as `[]` and a list node is written as `x : xs`. The value `x` is called the *head* and the rest of the list `xs` is called the *tail*. Thus:

```
"hi!" == ['h', 'i', '!'] == 'h' : ('i' : ('!' : []))
== 'h' : 'i' : '!' : []
```

When we define recursive functions on lists, we use the last form for pattern matching.

**Example**

(Re)-define the functions *length*, *take* and *drop*. 
Induction on Lists

If lists weren’t already defined in the standard library, we could define them ourselves:

\[
\text{data } \text{List } a = \text{Nil} \mid \text{Cons } a \ (\text{List } a)
\]

Induction

If we want to prove that a proposition holds for all lists:

\[
\forall xs. \ P(xs)
\]

It suffices to:

1. Show \( P([]) \) (the base case from nil)
2. Assuming the inductive hypothesis \( P(xs) \), show \( P(x:xs) \) (the inductive case from cons).
Induction on Lists

Example (Take and Drop)

- Show that \( \text{take} \ (\text{length} \ x) \ x = x \) for all \( x \).
- Show that \( \text{take} \ 5 \ x \cdot \cdot \cdot \text{drop} \ 5 \ x = x \) for all \( x \).
  - Sometimes we must generalise the proof goal.
  - Sometimes we must prove auxiliary lemmas.
Binary Trees

data Tree a = Leaf
   | Branch a (Tree a) (Tree a)

Induction Principle

To prove a property $P(t)$ for all trees $t$:

- Prove the base case $P(\text{Leaf})$.
- Assuming the two inductive hypotheses:
  - $P(l)$ and
  - $P(r)$

We must show $P(\text{Branch} \times l r)$.

Example (Tree functions)

Define leaves and height, and show $\forall t. \text{height } t < \text{leaves } t$
Rose Trees

data Forest a = Empty | Cons (Rose a) (Forest a)

data Rose a = Node a (Forest a)

Note that Forest and Rose are defined **mutually**.

**Example (Rose tree functions)**

Define *size* and *height*, and try to show

\[ \forall t. \text{height } t \leq \text{size } t \]
Simultaneous Induction

To prove a property about two types defined mutually, we have to prove two properties simultaneously.

```haskell
data Forest a = Empty | Cons (Rose a) (Forest a)
```

```haskell
data Rose a = Node a (Forest a)
```

**Inductive Principle**

To prove a property $P(t)$ about all $Rose$ trees $t$ and a property $Q(ts)$ about all $Forest$ $ts$ simultaneously:

- Prove $Q(Empty)$
- Assuming $P(t)$ and $Q(ts)$ (inductive hypotheses), show $Q(Cons t ts)$.
- Assuming $Q(ts)$ (inductive hypothesis), show $P(Node x ts)$. 