Binary Search Trees

(the sum, the product and the functor)

Oles Hodych

Melbourne Scala User Group

28 May 2018

Contents

- Sets and recursive definitions.
- Choosing the right abstraction.
- Product and sum types.
- Processing set elements, functoriality.
- A set for anything.

A declarative definition of a set

A *set* is an unordered collection of objects, called *elements* or *members* of a set.

A declarative definition of a set

A *set* is an unordered collection of objects, called *elements* or *members* of a set.

A set is said to *contain* its elements.

A declarative definition of a set

A *set* is an unordered collection of objects, called *elements* or *members* of a set.

A set is said to *contain* its elements.

The $a \in A$ notation denotes that a is an element of set A.

A declarative definition of a set

A *set* is an unordered collection of objects, called *elements* or *members* of a set.

A set is said to *contain* its elements.

The $a \in A$ notation denotes that a is an element of set A.

The $a \notin A$ notation denotes that a is not an element of set A.

Set properties

- Sets are *unordered* collections.

Set properties

- Sets are *unordered* collections.
- Sets contain no duplicate elements.

Problem statement

- Create a data abstraction for representing a finite set of elements.
- Implement set operations:
  - $adjoin(x, S)$ – produces a new set that has all elements of set S and element x;
  - $contains?(x, S)$ – a predicate that checks if $x \in S$;

Examples of sets

- $S = \{2,\ 1,\ 3,\ 4\}$
- $[a, b] = \{x \,|\, a \leqslant x \leqslant b\}$
- $\mathbb{N}_0 = \{0,\ 1,\ 2,\ \ldots\}$

Recursive definition of sets and other structures

Basis step:

Specify the initial collection of elements.

Recursive step:

Specify the rules for forming new elements from already present ones.

Recursive definition of sets, example

Basis step:

$$3 \in S.$$

Recursive step:

$$\text{if } x \in S \text{ and } y \in S \text{ then } (x + y) \in S$$

Recursive definition of sets, example

Basis step: $S_0 = \{3\}$

Recursive definition of sets, example

Basis step: $S_0 = \{3\}$

Recursive step 1: $S_1 = \{3, 6\}$ $3 + 3$

Recursive definition of sets, example

Basis step:	$S_0 = \{3\}$	
Recursive step 1:	$S_1 = \{3, 6\}$	$3 + 3$
Recursive step 2:	$S_2 = \{3, 6, 9, 12\}$	$3 + 6$ and $6 + 6$

Recursive definition of sets, example

Basis step:	$S_0 = \{3\}$	
Recursive step 1:	$S_1 = \{3, 6\}$	$3 + 3$
Recursive step 2:	$S_2 = \{3, 6, 9, 12\}$	$3 + 6$ and $6 + 6$
Recursive step 3:	$S_4 = \{3, 6, 9, 12, 15, 18, 21, 24\}$	$3+12$, $6+9$, $6+12$, $9+12$ and $12 + 12$

Recursive definition of sets, example

Basis step:	$S_0 = \{3\}$	
Recursive step 1:	$S_1 = \{3, 6\}$	$3 + 3$
Recursive step 2:	$S_2 = \{3, 6, 9, 12\}$	$3 + 6$ and $6 + 6$
Recursive step 3:	$S_4 = \{3, 6, 9, 12, 15, 18, 21, 24\}$	$3+12$, $6+9$, $6+12$, $9+12$ and $12 + 12$
...	...	

Structural induction

Basis step:

Show that statement P holds for instances that represents our structure at its basis step.

Inductive step:

Inductive hypothesis: P holds for all instances of out structure after applying k recursive steps.

Show that $P(x)$ holds for all new instances formed by applying the $k + 1$ recursive step.

Problem statement, simplified

- Create a data abstraction for representing a finite set of *integers*.
- Implement set operations:
  - $adjoin(x, S)$ – produces a new set that has all elements of set S and element x;
  - $contains?(x, S)$ – a predicate that checks if $x \in S$;

Sets as unordered lists

```scala
type IntSet = List[Int]

def contains_?(x: Int, set: IntSet): Boolean = ???

def adjoin(x: Int, set: IntSet): IntSet = ???
```

Sets as unordered lists

```
def contains_?(x: Int, set: IntSet): Boolean =
  if (set.isEmpty)
    false
  else
    x.equals(set.head) || contains_?(x, set.tail)
```

```
def adjoin(x: Int, set: IntSet): IntSet =
  if (contains_?(x, set))
    set
  else
    x::set
```

Sets as unordered lists

But what is the complexity of these operations?

Sets as unordered lists

But what is the complexity of these operations?

- Linear time inclusion (`adjoin`) of a new number into a set, $O(n)$.

Sets as unordered lists

But what is the complexity of these operations?

- Linear time inclusion (`adjoin`) of a new number into a set, $O(n)$.
- Linear time search to check (`contains?`) if a number is present in a set, $O(n)$.

Sets as unordered lists

But what is the complexity of these operations?

- Linear time inclusion (`adjoin`) of a new number into a set, $O(n)$.
- Linear time search to check (`contains?`) if a number is present in a set, $O(n)$.
- Linear space growth characteristics, $O(n)$.

Sets as binary search trees, 2-trees

The advantages of using a binary search tree as the data structure for implementing a set:

Sets as binary search trees, 2-trees

The advantages of using a binary search tree as the data structure for implementing a set:

- Logarithmic time inclusion (`adjoin`) of a new number into a set, $O(\lceil log_2 n \rceil)$.

Sets as binary search trees, 2-trees

The advantages of using a binary search tree as the data structure for implementing a set:

- Logarithmic time inclusion (`adjoin`) of a new number into a set, $O(\lceil log_2 n \rceil)$.
- Logarithmic time search to check (`contains?`) if a number is present in a set, $O(\lceil log_2 n \rceil)$.

Sets as binary search trees, 2-trees

The advantages of using a binary search tree as the data structure for implementing a set:

- Logarithmic time inclusion (`adjoin`) of a new number into a set, $O(\lceil log_2 n \rceil)$.
- Logarithmic time search to check (`contains?`) if a number is present in a set, $O(\lceil log_2 n \rceil)$.
- Linear space growth characteristics, $O(n)$.

$O(n)$ vs. $O(\lceil log_2 n \rceil)$

$$|S| = 15$$

$O(n)$ vs. $O(\lceil log_2 n \rceil)$

$|S| = 15$

15 vs. 4

$O(n)$ vs. $O(\lceil log_2 n \rceil)$

$|S| = 15$

15 vs. 4

$|S| = 1,000,000,000$

$O(n)$ vs. $O(\lceil log_2 n \rceil)$

$|S| = 15$

15 vs. 4

$|S| = 1,000,000,000$

$1,000,000,000$ vs. 30

Recursive definition of tree structures, full binary tree

Basis step:

A single node forms a full binary tree.

Recursive step:

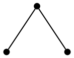
If T_1 and T_2 are disjoint full binary trees then there is a full binary tree $T_1 \cdot T_2$, which consists of a root node r together with edges connecting this root to the roots of the left subtree T_1 and the right subtree T_2.

Recursive definition of tree structures, full binary tree

Basis step: $\bullet$

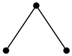
Recursive definition of tree structures, full binary tree

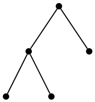
Basis step: 

Recursive step 1: 

Recursive definition of tree structures, full binary tree

Basis step: 

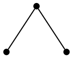
Recursive step 1: 

Recursive step 2: 

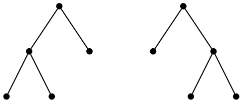
Recursive definition of tree structures, full binary tree

Basis step:

Recursive step 1:



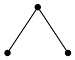
Recursive step 2:



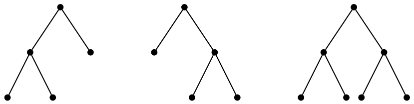
Recursive definition of tree structures, full binary tree

Basis step:

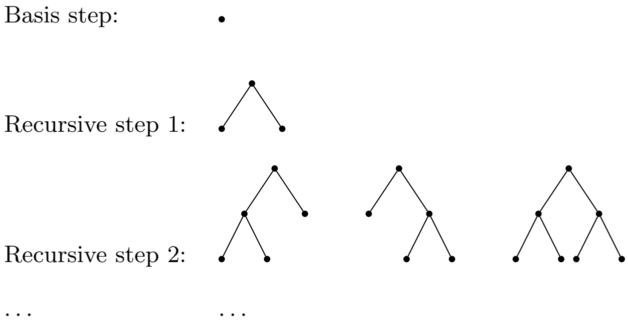
Recursive step 1:



Recursive step 2:

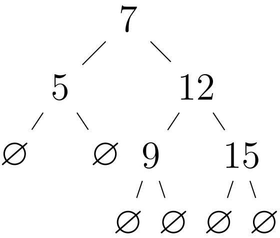


Recursive definition of tree structures, full binary tree



Full binary search tree

$$S = \{5, 7, 9, 12, 15\}$$



Recursively defined functions for recursively defined structures

Let's introduce some functions on full binary trees.

Recursively defined functions for recursively defined structures

Let's introduce some functions on full binary trees.

- $h : T_{FB} \to \mathbb{N}_0$ to compute the height of a full binary tree.
- $n : T_{FB} \to \mathbb{N}$ to compute the number of nodes in a full binary tree.

Recursive definition of $h(T)$

Basis step:

If T is a full binary tree consisting only of a root node then its height is 0.
That is $h(T) = 0$.

Recursive definition of $h(T)$

Basis step:

If T is a full binary tree consisting only of a root node then its height is 0.
That is $h(T) = 0$.

Recursive step:

If T_1 and T_2 are full binary trees then a full binary tree $T = T_1 \cdot T_2$ has the height of $h(T) = 1 + max(\ h(T_1),\ h(T_2)\ )$.

Recursive definition of $n(T)$

Basis step:

If T is a full binary tree consisting only of a root node then $n(T) = 1$.

Recursive definition of $n(T)$

Basis step:

If T is a full binary tree consisting only of a root node then $n(T) = 1$.

Recursive step:

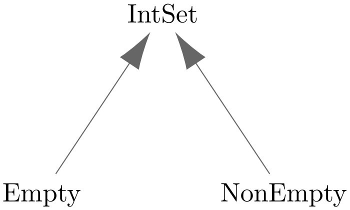
If T_1 and T_2 are full binary trees then a full binary tree $T = T_1 \cdot T_2$ has $n(T) = 1 + n(T_1) + n(T_2)$ nodes.

Recursively defined functions for recursively defined structures

Observation: functions on hierarchical recursively defined structures can only be defined recursively.

Hierarchical data by means of type hierarchies

Hierarchical data by means of type hierarchies



IntSet

```
type Set = IntSet

sealed trait IntSet {
    def contains_?(x: Int): Boolean

    def adjoin(x: Int): IntSet
}
```

Empty set

```
case object Empty extends IntSet {
    def contains_?(x: Int): Boolean =
        false

    def adjoin(x: Int): IntSet =
        NonEmpty(x, Empty, Empty)
}
```

Non empty set, operation `contains?`

```
case class NonEmpty(el: Int,
                    left: IntSet,
                    right: IntSet) extends IntSet {

    def contains_?(x: Int): Boolean =
        if (x < el) left contains_? x
        else if (x > el) right contains_? x
        else true

    def adjoin(x: Int): IntSet = ...
}
```

Non empty set, operation `adjoin`

```
case class NonEmpty(el: Int,
                    left: IntSet,
                    right: IntSet) extends IntSet {

  def contains_(x: Int): Boolean = ...

  def adjoin(x: Int): IntSet =
     if (x < el) NonEmpty(el, left adjoin x, right)
     else if (x > el) NonEmpty(el, left, right adjoin x)
     else this
}
```

IntSet in action

```
scala> val set = NonEmpty(7, Empty, Empty)
set: NonEmpty = NonEmpty(7,Empty,Empty)
```

IntSet in action

```
scala> val set = NonEmpty(7, Empty, Empty)
set: NonEmpty = NonEmpty(7,Empty,Empty)
```

```
scala> val set2 = set adjoin 5 adjoin 12
set2: NonEmpty = NonEmpty(7,NonEmpty(5,Empty,Empty),NonEmpty(12,E
```

Empty and non empty set, improving `toString`

```
case object Empty extends IntSet {
    ...
    override def toString = "."
}
```

Empty and non empty set, improving toString

```
case object Empty extends IntSet {
    ...
    override def toString = "."
}
```

```
case class NonEmpty(el: Int,
                    left: IntSet,
                    right: IntSet) extends IntSet {
    ...
    override def toString =
        "{" + left + el + right + "}"
}
```

IntSet back in action

```
scala> val set = NonEmpty(7, Empty, Empty)
set: NonEmpty = {.7.}
```

IntSet back in action

```
scala> val set = NonEmpty(7, Empty, Empty)
set: NonEmpty = {.7.}
```

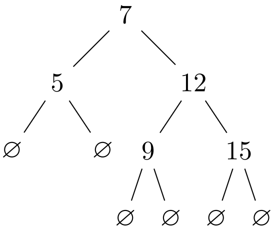
```
scala> val set2 = set adjoin 5 adjoin 12
set2: IntSet = {{.5.}7{.12.}}
```

Persistent data structures, analysing operation `adjoin`

```
def adjoin(x: Int): IntSet = NonEmpty(x, Empty, Empty)

def adjoin(x: Int): IntSet =
    if (x < el) NonEmpty(el, left adjoin x, right)
    else if (x > el) NonEmpty(el, left, right adjoin x)
    else this
}

set adjoin 3
```

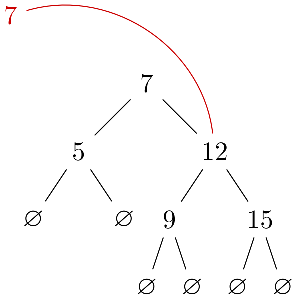


Persistent data structures, analysing operation `adjoin`

```
def adjoin(x: Int): IntSet = NonEmpty(x, Empty, Empty)

def adjoin(x: Int): IntSet =
    if (x < el) NonEmpty(el, left adjoin x, right)
    else if (x > el) NonEmpty(el, left, right adjoin x)
    else this
}

set adjoin 3
```

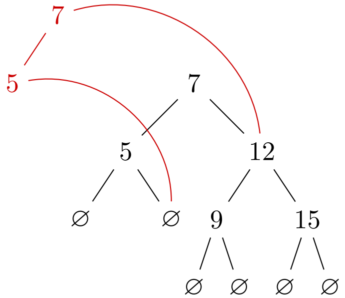


Persistent data structures, analysing operation `adjoin`

```
def adjoin(x: Int): IntSet = NonEmpty(x, Empty, Empty)

def adjoin(x: Int): IntSet =
   if (x < el) NonEmpty(el, left adjoin x, right)
   else if (x > el) NonEmpty(el, left, right adjoin x)
   else this
}

set adjoin 3
```

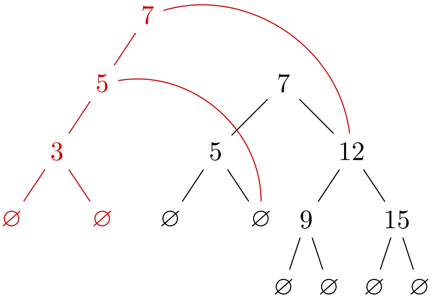


Persistent data structures, analysing operation `adjoin`

```
def adjoin(x: Int): IntSet = NonEmpty(x, Empty, Empty)

def adjoin(x: Int): IntSet =
    if (x < el) NonEmpty(el, left adjoin x, right)
    else if (x > el) NonEmpty(el, left, right adjoin x)
    else this
}

set adjoin 3
```



Implementing $h(T)$ and $n(T)$

Basis step:

$$h(T) = 0$$

Recursive step:

$$h(T) = 1 + max(\ h(T_1),\ h(T_2)\ )$$

Implementing $h(T)$ and $n(T)$

Basis step:

$$h(T) = 0$$

Recursive step:

$$h(T) = 1 + max(\ h(T_1),\ h(T_2)\ )$$

Basis step:

$$n(T) = 1$$

Recursive step:

$$n(T) = 1 + n(T_1) + n(T_2)$$

Which subtype represents an empty set?

```
sealed trait IntSet {
    ...
    def isEmpty: Boolean
}

case object Empty extends IntSet {
    ...
    def isEmpty: Boolean = true
}

case class NonEmpty(...) extends IntSet {
    ...
    def isEmpty: Boolean = false
}
```

Cartesian product and Product type

Cartesian product (Set Theory):

$$A = \{1, 3\},\ B = \{1, 2\}$$

Cartesian product and Product type

Cartesian product (Set Theory):

$$A = \{1, 3\},\ B = \{1, 2\}$$

$$A \times B = \{(1, 1),\ (1, 2)\ (3, 1),\ (3, 2)\}$$

Cartesian product and Product type

Cartesian product (Set Theory):

$$A = \{1, 3\},\ B = \{1, 2\}$$

$$A \times B = \{(1, 1),\ (1, 2)\ (3, 1),\ (3, 2)\}$$

Product type (Type Theory):

$$A \times B$$

Cartesian product and Product type

Cartesian product (Set Theory):

$$A = \{1, 3\},\ B = \{1, 2\}$$

$$A \times B = \{(1, 1),\ (1, 2)\ (3, 1),\ (3, 2)\}$$

Product type (Type Theory):

$$A \times B$$

$$\texttt{Rational} = \texttt{Int} \times \texttt{Int}$$

Cartesian product and Product type

Cartesian product (Set Theory):

$$A = \{1, 3\},\ B = \{1, 2\}$$

$$A \times B = \{(1, 1),\ (1, 2)\ (3, 1),\ (3, 2)\}$$

Product type (Type Theory):

$$A \times B$$

$$\texttt{Rational} = \texttt{Int} \times \texttt{Int}$$

$$\texttt{NonEmpty} = \texttt{Int} \times \texttt{IntSet} \times \texttt{IntSet}$$

Disjoint union and Sum type

Union (Set Theory):

$A = \{1, 3\}$, $B = \{1, 2\}$

Disjoint union and Sum type

Union (Set Theory):

$$A = \{1, 3\},\ B = \{1, 2\}$$

$$A \bigcup B = \{1,\ 3,\ 2\}$$

Disjoint union and Sum type

Union (Set Theory):

$$A = \{1, 3\},\ B = \{1, 2\}$$

$$A \bigcup B = \{1,\ 3,\ 2\}$$

Disjoint union or tagged union (Set Theory):

$$A = \{1, 3\},\ B = \{1, 2\}$$

Disjoint union and Sum type

Union (Set Theory):

$$A = \{1, 3\},\ B = \{1, 2\}$$

$$A \bigcup B = \{1,\ 3,\ 2\}$$

Disjoint union or tagged union (Set Theory):

$$A = \{1, 3\},\ B = \{1, 2\}$$

$$A^{*} = \{(1, a),\ (3, a)\},\ B^{*} = \{(1, b),\ (2, b)\}$$

Disjoint union and Sum type

Union (Set Theory):

$A = \{1,3\},\ B = \{1,2\}$

$A \bigcup B = \{1,\ 3,\ 2\}$

Disjoint union or tagged union (Set Theory):

$A = \{1,3\},\ B = \{1,2\}$

$A^{*} = \{(1,a),\ (3,a)\},\ B^{*} = \{(1,b),\ (2,b)\}$

$A \coprod B = A^{*} \bigcup B^{*} = \{(1,a),\ (3,a)\ (1,b),\ (2,b)\}$

Disjoint union and Sum type

Union (Set Theory):

$A = \{1, 3\}$, $B = \{1, 2\}$

$A \bigcup B = \{1,\ 3,\ 2\}$

Disjoint union or tagged union (Set Theory):

$A = \{1, 3\}$, $B = \{1, 2\}$

$A^{*} = \{(1, a),\ (3, a)\}$, $B^{*} = \{(1, b),\ (2, b)\}$

$A \coprod B = A^{*} \bigcup B^{*} = \{(1, a),\ (3, a)\ (1, b),\ (2, b)\}$

Sum type (Type Theory):

$A + B$

Disjoint union and Sum type

Union (Set Theory):

$$A = \{1, 3\},\ B = \{1, 2\}$$

$$A \bigcup B = \{1,\ 3,\ 2\}$$

Disjoint union or tagged union (Set Theory):

$$A = \{1, 3\},\ B = \{1, 2\}$$

$$A^{*} = \{(1, a),\ (3, a)\},\ B^{*} = \{(1, b),\ (2, b)\}$$

$$A \coprod B = A^{*} \bigcup B^{*} = \{(1, a),\ (3, a)\ (1, b),\ (2, b)\}$$

Sum type (Type Theory):

$$A + B$$

$\texttt{IntSet} = \texttt{Empty} + \texttt{NonEmpty}$

Disjoint union and Sum type

Union (Set Theory):

$A = \{1, 3\}$, $B = \{1, 2\}$

$A \bigcup B = \{1,\ 3,\ 2\}$

Disjoint union or tagged union (Set Theory):

$A = \{1, 3\}$, $B = \{1, 2\}$

$A^* = \{(1, a),\ (3, a)\}$, $B^* = \{(1, b),\ (2, b)\}$

$A \coprod B = A^* \bigcup B^* = \{(1, a),\ (3, a)\ (1, b),\ (2, b)\}$

Sum type (Type Theory):

$A + B$

$\texttt{IntSet} = \texttt{Empty} + \texttt{NonEmpty}$

$\texttt{IntSet} = \texttt{Empty} \mid \texttt{NonEmpty(0, Empty, Empty)} \mid \ldots$

Disjoint union and Sum type

Union (Set Theory):

$A = \{1, 3\}$, $B = \{1, 2\}$

$A \bigcup B = \{1,\ 3,\ 2\}$

Disjoint union or tagged union (Set Theory):

$A = \{1, 3\}$, $B = \{1, 2\}$

$A^* = \{(1, a),\ (3, a)\}$, $B^* = \{(1, b),\ (2, b)\}$

$A \coprod B = A^* \bigcup B^* = \{(1, a),\ (3, a)\ (1, b),\ (2, b)\}$

Sum type (Type Theory):

$A + B$

$\texttt{IntSet} = \texttt{Empty} + \texttt{NonEmpty}$

$\texttt{IntSet} = \texttt{Empty} \mid \texttt{NonEmpty(0, Empty, Empty)} \mid \ldots$

$\texttt{Boolean} = \texttt{true} \mid \texttt{false}$

Disjoint union and Sum type

Union (Set Theory):

$$A = \{1, 3\},\ B = \{1, 2\}$$

$$A \bigcup B = \{1,\ 3,\ 2\}$$

Disjoint union or tagged union (Set Theory):

$$A = \{1, 3\},\ B = \{1, 2\}$$

$$A^* = \{(1, a),\ (3, a)\},\ B^* = \{(1, b),\ (2, b)\}$$

$$A \coprod B = A^* \bigcup B^* = \{(1, a),\ (3, a)\ (1, b),\ (2, b)\}$$

Sum type (Type Theory):

$$A + B$$

$$\texttt{IntSet} = \texttt{Empty} + \texttt{NonEmpty}$$

$$\texttt{IntSet} = \texttt{Empty} \mid \texttt{NonEmpty(0, Empty, Empty)} \mid \ldots$$

$$\texttt{Boolean} = \texttt{true} \mid \texttt{false}$$

$$\texttt{Int} = 1 \ \mid\ 2 \ \mid\ 3 \ \mid\ \ldots$$

Sealed type hierarchies for Sum and Prod types

```scala
sealed trait IntSet {
    ...
}

case object Empty extends IntSet {
    ...
}

case class NonEmpty(el: Int
                    left: IntSet,
                    right: IntSet) extends IntSet {
    ...
}
```

Decomposition of compound data, pattern matching

```
e match {
    case p1 => e1
    case p2 => e2
    ...
    case pn => en
}
```

http://www.scala-lang.org/files/archive/spec/2.12/08-pattern-matching.html

Implementing $h(T)$ with pattern matching

```
def height(set: IntSet): Int = set match {
    case Empty =>
        0
    case NonEmpty(_, left, right) =>
        1 + Math.max(height(left), height(right))
}
```

Implementing $n(T)$ with pattern matching

```scala
def size(set: IntSet): Int = set match {
    case Empty =>
        0
    case NonEmpty(_, left, right) =>
        1 + size(left) + size(right)
}
```

Processing elements in sets

How do we go about doing something with elements in our `IntSet`?

Double each element

```
def double(set: IntSet): IntSet = set match {
    case Empty =>
        set
    case NonEmpty(el, left, right) =>
        NonEmpty(2 * el, double(left), double(right))
}
```

Double each element in action

```
scala> val set = Empty include 7 include 5 include (
                 12) include 9 include 15
set: IntSet = {{.5.}7{{.9.}12{.15.}}}

scala> double(set)
res3: IntSet = {{.10.}14{{.18.}24{.30.}}}
```

Square each element

```
def square(set: IntSet): IntSet = set match {
    case Empty =>
        set
    case NonEmpty(el, left, right) =>
        NonEmpty(el * el, square(left), square(right))
}
```

Square each element in action

```
scala> val set = Empty include 7 include 5 include (
            12) include 9 include 15
set: IntSet = {{.5.}7{{.9.}12{.15.}}}

scala> square(set)
res6: IntSet = {{.25.}49{{.81.}144{.225.}}}
```

Processing elements in sets

Can processing of elements in `IntSet` be generalised?

Mapping elements

```
sealed trait IntSet {
    def contains_?(x: Int): Boolean
    def adjoin(x: Int): IntSet

    def map(f: Int => Int): IntSet = this match {
      case Empty =>
        this
      case NonEmpty(el, left, right) =>
        NonEmpty(f(el), left map f, right map f)
    }
}
```

Mapping elements in action

```
scala> val set = Empty include 7 include 5 include (
                12) include 9 include 15
set: IntSet = {{.5.}7{{.9.}12{.15.}}}

scala> set map (x=>2*x)
res10: IntSet = {{.10.}14{{.18.}24{.30.}}}

scala> set map (x=>x*x)
res11: IntSet = {{.25.}49{{.81.}144{.225.}}}
```

Mapping elements in action, alternative syntax

```
scala> set map { x=>2*x }
res12: IntSet = {{.10.}14{{.18.}24{.30.}}}

scala> set map { 2*_ }
res13: IntSet = {{.10.}14{{.18.}24{.30.}}}
```

Mapping is not easy...

The presented implementation of `map` is invalid!

Mapping is not easy...

The presented implementation of `map` is invalid!

```
scala> val set = Empty include 7 include 5 ... include 15
set: IntSet = {{.5.}7{{.9.}12{.15.}}}

scala> set map { x => if (x > 7) -x else x }
res14: IntSet = {{.5.}7{{.-15{.-12.}}-9.}}
```

Mapping is not easy...

The presented implementation of `map` is invalid!

```
scala> val set = Empty include 7 include 5 ... include 15
set: IntSet = {{.5.}7{{.9.}12{.15.}}}

scala> set map { x => if (x > 7) -x else x }
res14: IntSet = {{.5.}7{{.-15{.-12.}}-9.}}
```

```
scala> set map { x => 1 }
res14: IntSet = {{.1.}1{{.1.}1{.1.}}}
```

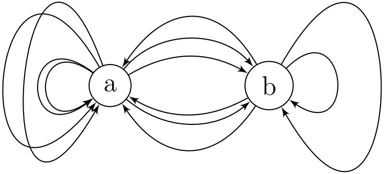
Category theory, simplistic view

Major tools in the categorical toolbox:

- **Abstraction**
- **Composition**
- **Identity**

Category theory, intuitive definition

A category is a bunch of objects with morphisms between them.



Category theory, intuitive definition

- **Objects** – are primitives, have no structure or properties.
- **Arrows** – morphisms, are primitives, join objects, have the beginning and the end.

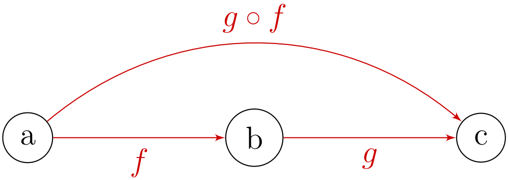
Category theory, intuitive definition

- Composition – arrows always compose.

$$
\begin{tikzcd}
a \arrow[r, "f"'] & b \arrow[r, "g"'] & c
\end{tikzcd}
$$

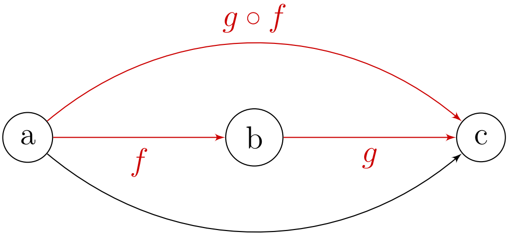
Category theory, intuitive definition

- Composition – arrows always compose.



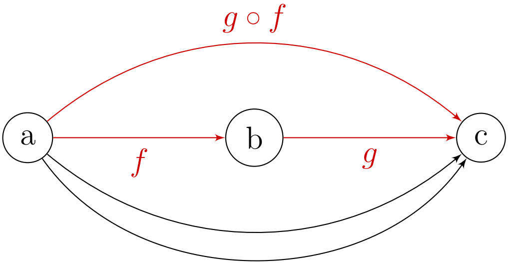
Category theory, intuitive definition

- Composition – arrows always compose.



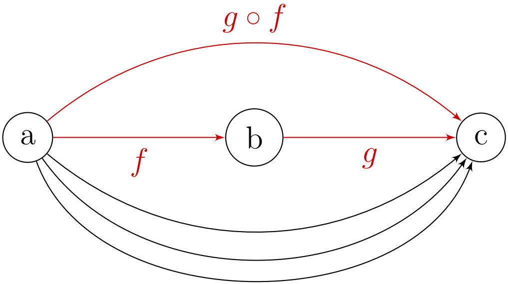
Category theory, intuitive definition

- Composition – arrows always compose.



Category theory, intuitive definition

- Composition – arrows always compose.



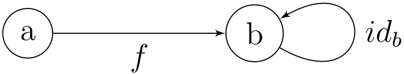
Category theory, intuitive definition

- Identity – for every object there is an identity arrow.

$$\begin{tikzcd} a \arrow[loop left, "id_a"] \arrow[r, "f"'] & b \arrow[loop right, "id_b"] \end{tikzcd}$$

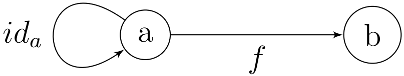
Category theory, axioms

- Left identity $id_a \circ f = f$



Category theory, axioms

- Right identity $f \circ id_a = f$



Category theory, axioms

- Associativity $h \circ (g \circ f) = (h \circ g) \circ f$

$$a \xrightarrow{f} b \xrightarrow{g} c \xrightarrow{h} d$$

Category theory, axioms

- Associativity $h \circ (g \circ f) = (h \circ g) \circ f$

$$\begin{tikzcd} a \arrow[r, "f"'] \arrow[rr, bend left, "g \circ f"] & b \arrow[r, "g"'] & c \arrow[r, "h"'] & d \end{tikzcd}$$

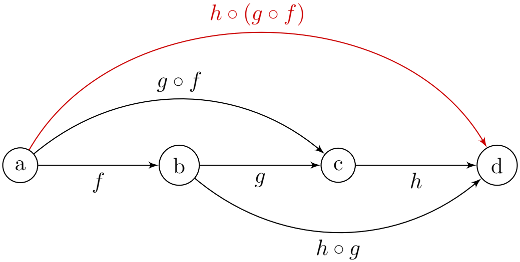
Category theory, axioms

- Associativity $h \circ (g \circ f) = (h \circ g) \circ f$

$$\begin{tikzcd}
a \arrow[r, "f"'] \arrow[rr, bend left, "g \circ f"] \arrow[rrr, bend left=60, color=red, "h \circ (g \circ f)"] & b \arrow[r, "g"'] & c \arrow[r, "h"'] & d
\end{tikzcd}$$

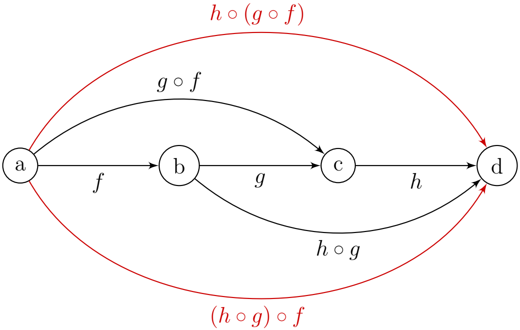
Category theory, axioms

- Associativity $h \circ (g \circ f) = (h \circ g) \circ f$



Category theory, axioms

- Associativity $h \circ (g \circ f) = (h \circ g) \circ f$



Category theory and programming

- Objects are types.
- Arrows are functions.

Morphisms, intuition

A **morphism** is a structure-preserving map from one mathematical object to another.

Homomorphisms, intuition

A **homomorphism** is a structure-preserving map between two algebraic structures of the same type.

Functors, intuition

A **functor** is a *homomorphism* between categories in the category theory.

Functors, intuition

What defines the structure in a category?

Functors, intuition

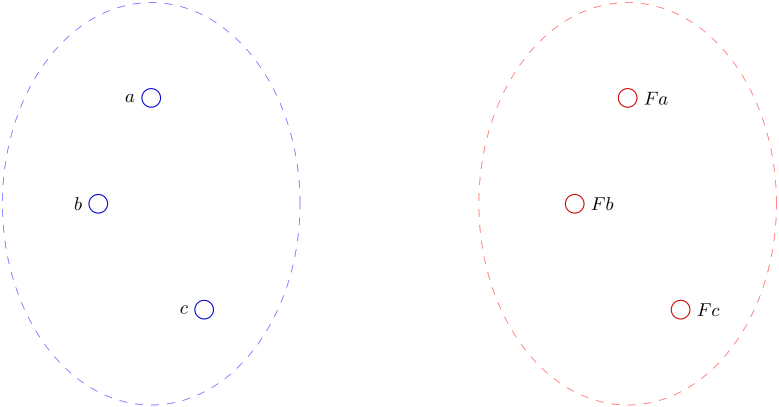
What defines the structure in a category?

Arrows and their composition!

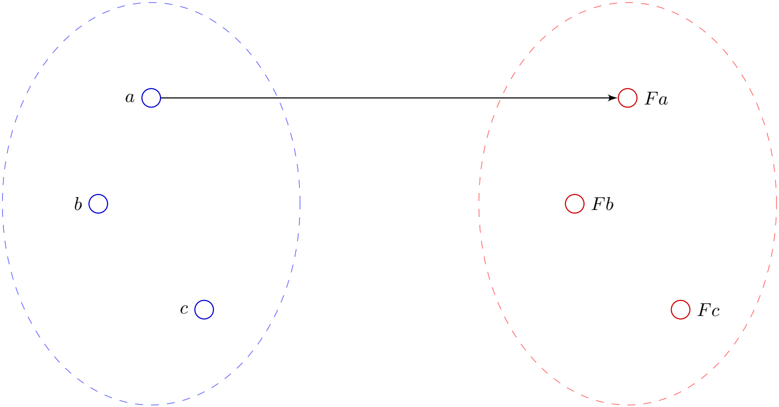
Functors, intuition

Functors must map objects to objects and arrows to arrows while preserving composition!

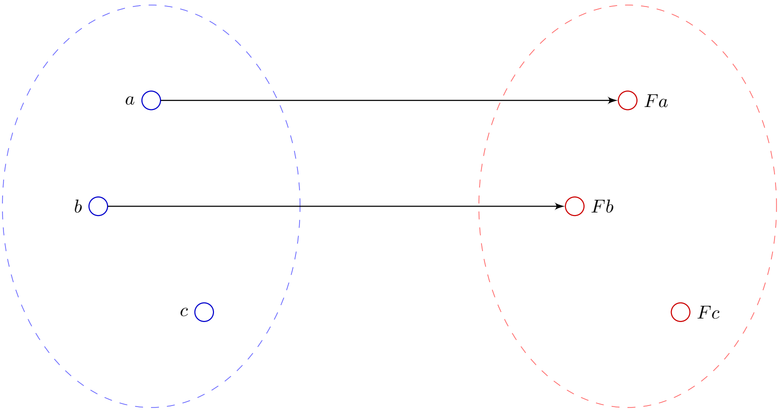
Functors, preserving structure



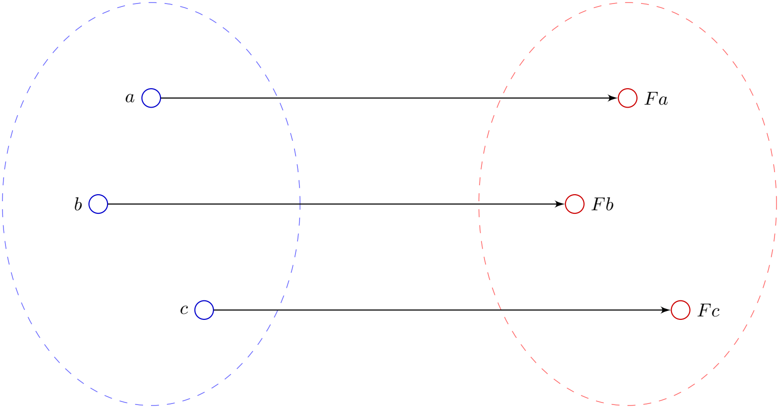
Functors, preserving structure



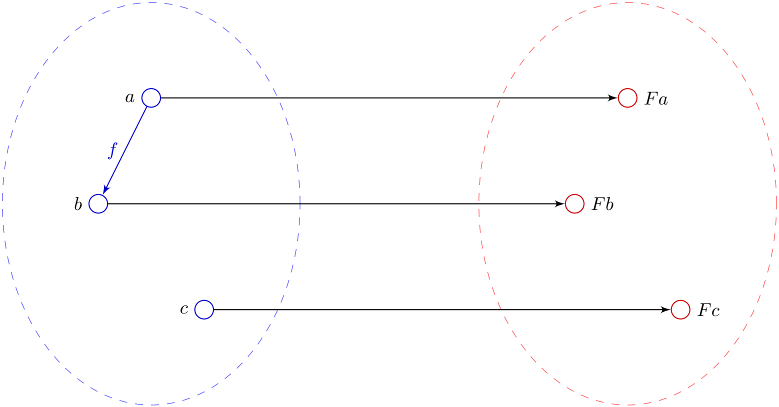
Functors, preserving structure



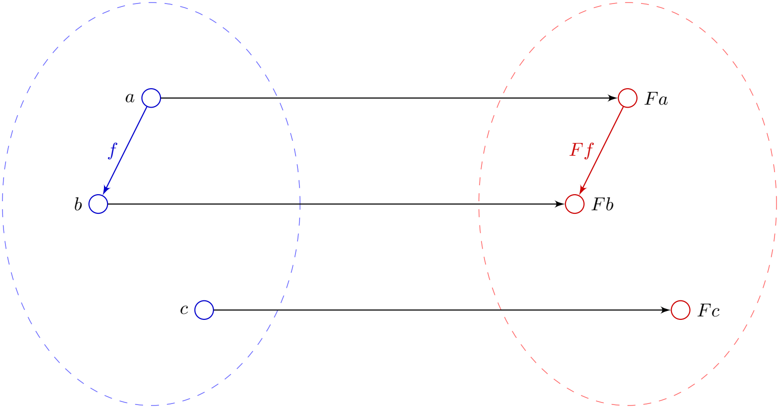
Functors, preserving structure



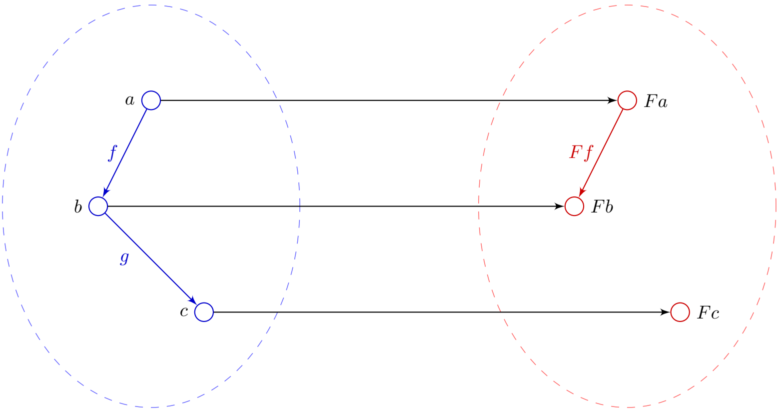
Functors, preserving structure



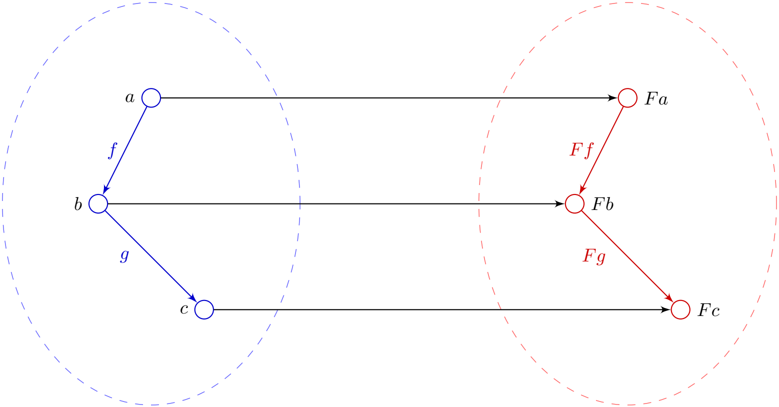
Functors, preserving structure



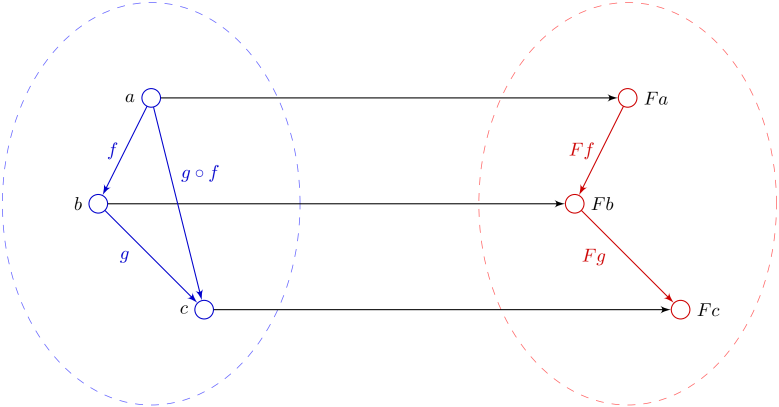
Functors, preserving structure



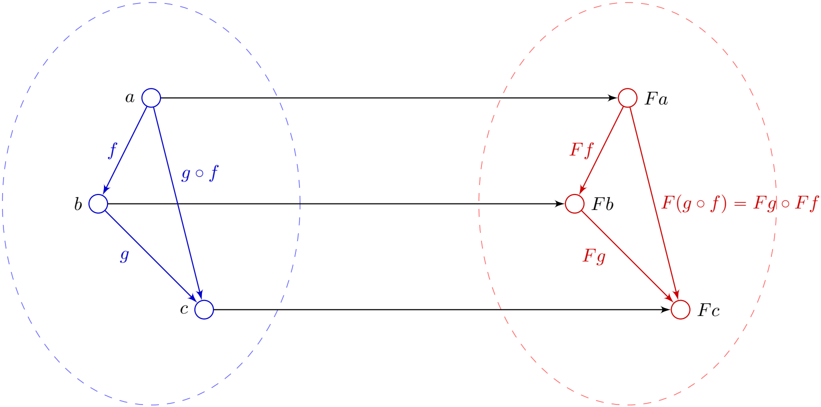
Functors, preserving structure



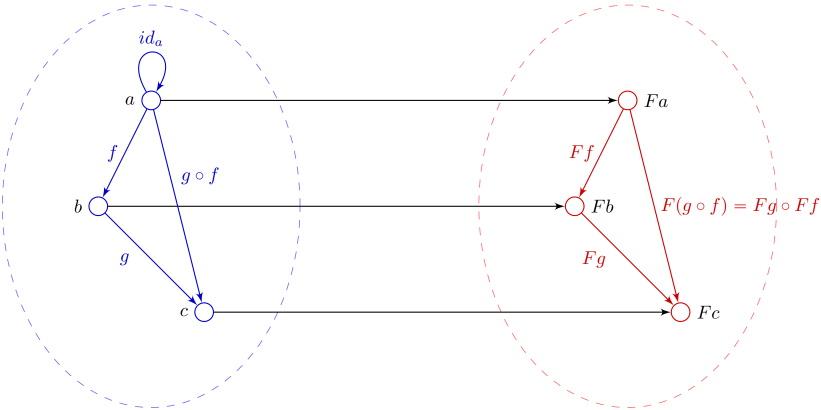
Functors, preserving structure



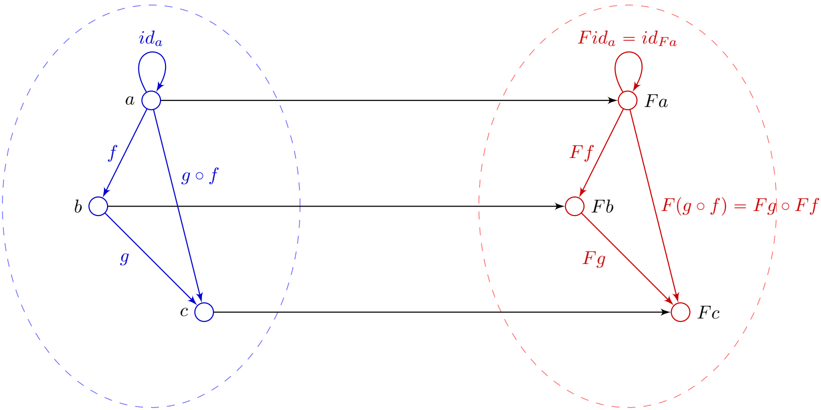
Functors, preserving structure



Functors, preserving structure



Functors, preserving structure



Functors, further intuition

A Functor is a collection of many morphisms for mapping all objects and all arrows.

Functors, programming

Objects are **types**, arrows are **functions** between types.

Functors, programming

Objects are **types**, arrows are **functions** between types.

Functors are mappings between types and functions.

Functors, `Int` and `IntSet`

Functors, `Int` and `IntSet`

$$Int \longrightarrow IntSet$$

Functors, `Int` and `IntSet`

Functors, `Int` and `IntSet`

Functors, Int and IntSet, wishful thinking

```
scala> val set = Empty include 7 include 5 ... include 15
set: IntSet = {{.5.}7{{.9.}12{.15.}}}

scala> set map {x => x}
res23: IntSet = {{.5.}7{{.9.}12{.15.}}}
```

Functors, Int and IntSet, wishful thinking

```
scala> val set = Empty include 7 include 5 ... include 15
set: IntSet = {{.5.}7{{.9.}12{.15.}}}

scala> set map {x => x}
res23: IntSet = {{.5.}7{{.9.}12{.15.}}}
```

```
scala> set map {x => 1}
res24: IntSet = {.1.}
```

Making IntSet functorial

```scala
sealed trait IntSet {
  ...
  def map(f: Int => Int): IntSet = this match {
    case Empty =>
      this
    case NonEmpty(el, left, right) =>
      ???
  }
}
```

Making IntSet functorial

```
sealed trait IntSet {
  ...
  def map(f: Int => Int): IntSet = this match {
    case Empty =>
      this
    case NonEmpty(el, left, right) => {
      val l = left map f
      val v = f(el)
      val r = right map f
      l union r adjoin v
    }
  }
}
```

Set operation `union`

$union(S_1, S_2)$ – produces a new set that contains all elements from S_1 and S_2.

Set operation `union`

$union(S_1, S_2)$ – produces a new set that contains all elements from S_1 and S_2.

```
sealed trait IntSet {
    ...
    def union(other: IntSet): IntSet
}

case object Empty extends IntSet {
    ...
    def union(other: IntSet): IntSet = other
}

case class NonEmpty(...) extends IntSet {
    ...
    def union(other: IntSet): IntSet = ???
}
```

Set operation `union`

$union(S_1, S_2)$ – produces a new set that contains all elements from S_1 and S_2.

Set operation `union`

$union(S_1, S_2)$ – produces a new set that contains all elements from S_1 and S_2.

```
case class NonEmpty(el: Int,
                    left: IntSet,
                    right: IntSet) extends IntSet {
  ...
  def union(other: IntSet): IntSet =
    ((left union right) union other) adjoin el
}
```

Properties of a functor

Properties of a functor

- `set map {x => x} == set`

Properties of a functor

- `set map {x => x} == set`
- `set.map(g).map(f) == set map {f.compose(g)}`

Functors, functoriality

`IntSet` is functorial!

Generic sets

```
type Set = IntSet

trait IntSet {...}
```

Generic sets

```
type Set = IntSet

trait IntSet {...}
```

```
type Set[+A] = Tree[A]

trait Tree[+A] {...}
```

Generic sets, type hierarchy

```
type Set[+A] = Tree[A]

sealed trait Tree[+A] {
}

case object Empty extends Tree[Nothing] {
  override def toString = "."
}

case class NonEmpty[A](a: A,
                       left: Tree[A],
                       right: Tree[A]) extends Tree[A] {
  override def toString: String =
    "{" + left + a + right + "}"
}
```

Generic sets, operations

```scala
def adjoin[A](x: A, set: Set[A])
             (implicit cmp: Ordering[A]): Set[A] = ???

def contains_?[A](x: A, set: Set[A])
                 (implicit cmp: Ordering[A]): Boolean = ???

def union[A](set1: Set[A], set2: Set[A])
            (implicit cmp: Ordering[A]): Set[A] = ???

def map[A, B](set: Set[A], f: A => B)
             (implicit cmp: Ordering[B]): Set[B] = ???
```

Generic sets, operation `adjoin`

```scala
def adjoin[A](x: A, set: Set[A])
             (implicit cmp: Ordering[A]): Set[A] =
  set match {
  case Empty =>
    NonEmpty(x, Empty, Empty)
  case NonEmpty(a, left, right) =>
    if (cmp.lt(x, a)) NonEmpty(a, adjoin(x, left), right)
    else if (cmp.gt(x, a)) NonEmpty(a, left, adjoin(x, right))
    else set
}
```

Generic sets, operation contains?

```
def contains_?[A](x: A, set: Set[A])
                 (implicit cmp: Ordering[A]): Boolean =
  set match {
  case Empty =>
    false
  case NonEmpty(a, left, right) =>
    if (cmp.lt(x, a)) contains_?(x, left)
    else if (cmp.gt(x, a)) contains_?(x, right)
    else true
}
```

Generic sets, operation `union`

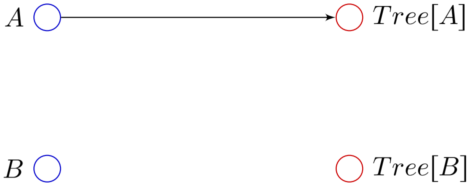
```
def union[A](set1: Set[A], set2: Set[A])
             (implicit cmp: Ordering[A]): Set[A] =
  set1 match {
  case Empty =>
    set2
  case NonEmpty(x, left, right) =>
    set2 match {
    case Empty =>
      set1
    case NonEmpty(_, _, _) =>
      adjoin(x, union(union(left, right), set2))
  }
}
```

Generic sets, operation map

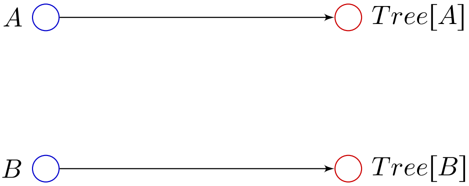
```scala
def map[A, B](set: Set[A], f: A => B)
             (implicit cmp: Ordering[B]): Set[B] =
  set match {
  case Empty =>
    Empty
  case NonEmpty(x, left, right) =>
    val l = map(left, f)
    val v = f(x)
    val r = map(right, f)
    adjoin(v, union(l, r))
}
```

Functoriality of `Tree[A]`

Functoriality of `Tree[A]`



Functoriality of `Tree[A]`



Functoriality of `Tree[A]`

$$
\begin{tikzcd}
A \arrow[r] \arrow[d, "f"'] & Tree[A] \\
B \arrow[r] & Tree[B]
\end{tikzcd}
$$

Functoriality of `Tree[A]`

$$
\begin{tikzcd}
A \arrow[r] \arrow[d, "f"'] & Tree[A] \arrow[d, "map\ f"] \\
B \arrow[r] & Tree[B]
\end{tikzcd}
$$

Functoriality of `Tree[A]`

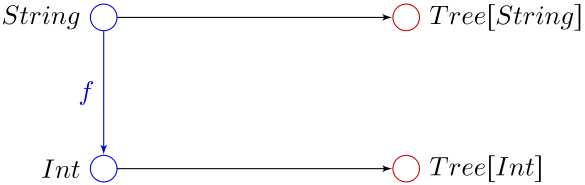
Functoriality of `Tree[A]`

$$
\begin{tikzcd}
String \arrow[rr] & & Tree[String] \\
Int & & Tree[Int]
\end{tikzcd}
$$

Functoriality of `Tree[A]`

$$\begin{tikzcd}
String \arrow[r] & Tree[String] \\
Int \arrow[r] & Tree[Int]
\end{tikzcd}$$

Functoriality of `Tree[A]`



Functoriality of `Tree[A]`

$$
\begin{tikzcd}
String \arrow[r] \arrow[d, "f"'] & Tree[String] \arrow[d, "map\ f"] \\
Int \arrow[r] & Tree[Int]
\end{tikzcd}
$$

A bit of coding. . .

Let's run some code!

Some final thoughts on binary search trees

Full binary search trees become unbalanced over time, and may even degenerate into a list.

Some final thoughts on binary search trees

Full binary search trees become unbalanced over time, and may even degenerate into a list.

- They require maintenance – rebalancing.

Some final thoughts on binary search trees

Full binary search trees become unbalanced over time, and may even degenerate into a list.

- They require maintenance – rebalancing.
- There are more efficient structures: B-trees and red-black trees.

Looking back

What have we touched on today?

Looking back

What have we touched on today?

- The representation problem.

Looking back

What have we touched on today?

- The representation problem.
- ADTs and how `sealed` type hierarchies of `trait`s, `case object`s and `case class`es can model them.

Looking back

What have we touched on today?

- The representation problem.
- ADTs and how `sealed` type hierarchies of `trait`s, `case object`s and `case class`es can model them.
- Persistent data structures.

Looking back

What have we touched on today?

- The representation problem.
- ADTs and how `sealed` type hierarchies of `trait`s, `case object`s and `case class`es can model them.
- Persistent data structures.
- Functoriality and what it means.

Looking back

What have we touched on today?

- The representation problem.
- ADTs and how `sealed` type hierarchies of `trait`s, `case object`s and `case class`es can model them.
- Persistent data structures.
- Functoriality and what it means.
- Used both more OO and more functional approach to modelling data structures and their operations.

Looking back

What have we touched on today?

- The representation problem.
- ADTs and how `sealed` type hierarchies of `traits`, `case objects` and `case classes` can model them.
- Persistent data structures.
- Functoriality and what it means.
- Used both more OO and more functional approach to modelling data structures and their operations.
- Scala `objects` as modules.
- `Implicit` function arguments.

Q&A