Principles of Programming Languages (Lecture 10)

COMP 3031, Fall 2025

Lionel Parreaux

Monads

Monads

Data structures with `map` and `flatMap` seem to be quite common.

In fact there's a name that describes such a class of a data structures together with some algebraic laws that they should have.

They are called *monads*.

What is a Monad?

A monad `M` is a parametric type `M[T]` with two operations, `flatMap` and `unit`, that have to satisfy some laws.

```
extension [T](m: M[T])
  def flatMap[U](f: T => M[U]): M[U]

def unit[T](x: T): M[T]
```

In the literature, `flatMap` is also called `bind`. It can be an extension method, or be defined as a regular method in the monad class `M`.

Examples of Monads

- `List` is a monad with `unit(x) = List(x)`
- `Set` is monad with `unit(x) = Set(x)`
- `Option` is a monad with `unit(x) = Some(x)`
- `Generator` is a monad with `unit(x) = single(x)`

With, in each case, the `flatMap` provided as a method of the type.

Examples of Monads

- `List` is a monad with `unit(x) = List(x)`
- `Set` is monad with `unit(x) = Set(x)`
- `Option` is a monad with `unit(x) = Some(x)`
- `Generator` is a monad with `unit(x) = single(x)`

With, in each case, the `flatMap` provided as a method of the type.

Q: What about `map`?

Monads and map

`map` can be defined for every monad as a combination of `flatMap` and `unit`:

```
m.map(f)  ==  m.flatMap(x => unit(f(x)))
          ==  m.flatMap(f andThen unit)
```

Note: `andThen` is defined function composition in the standard library.

```
extension [A, B](f: A => B)
  infix def andThen[C](g: B => C): A => C =
    x => g(f(x))
```

Monad Laws

To qualify as a monad, a type has to satisfy three laws:

Associativity:

```
m.flatMap(f).flatMap(g)  ==  m.flatMap(x => f(x).flatMap(g))
```

Left unit

```
unit(x).flatMap(f)  ==  f(x)
```

Right unit

```
m.flatMap(unit)  ==  m
```

Checking Monad Laws

Let's check the monad laws for Option.

Here's `flatMap` for `Option`:

```
extension [T](xo: Option[T])
  def flatMap[U](f: T => Option[U]): Option[U] = xo match
    case Some(x) => f(x)
    case None => None
```

Checking the Left Unit Law

Need to show: `Some(x).flatMap(f)  ==  f(x)`

```
Some(x).flatMap(f)
```

Checking the Left Unit Law

Need to show: `Some(x).flatMap(f)  ==  f(x)`

```
      Some(x).flatMap(f)

==    Some(x) match
        case Some(x) => f(x)
        case None => None
```

Checking the Left Unit Law

Need to show: `Some(x).flatMap(f)  ==  f(x)`

```
       Some(x).flatMap(f)

==     Some(x) match
         case Some(x) => f(x)
         case None => None

==     f(x)
```

Checking the Right Unit Law

Need to show: `opt.flatMap(Some(_))  ==  opt`

```
opt.flatMap(Some(_))
```

Checking the Right Unit Law

Need to show: `opt.flatMap(Some(_))  ==  opt`

```
      opt.flatMap(Some(_))

==    opt match
        case Some(x) => Some(x)
        case None => None
```

Checking the Right Unit Law

Need to show: `opt.flatMap(Some(_))  ==  opt`

```
      opt.flatMap(Some(_))

==    opt match
        case Some(x) => Some(x)
        case None => None

==    opt
```

Checking the Associative Law

Need to show:

```
opt.flatMap(f).flatMap(g)  ==  opt.flatMap(x => f(x).flatMap(g))
```

```
    opt.flatMap(f).flatMap(g)
```

Checking the Associative Law

Need to show:

```
opt.flatMap(f).flatMap(g)  ==  opt.flatMap(x => f(x).flatMap(g))
```

```
       opt.flatMap(f).flatMap(g)

  ==   (opt match { case Some(x) => f(x) case None => None })
            match { case Some(y) => g(y) case None => None }
```

Checking the Associative Law

Need to show:

```
opt.flatMap(f).flatMap(g)  ==  opt.flatMap(x => f(x).flatMap(g))
```

```
       opt.flatMap(f).flatMap(g)

  ==   (opt match { case Some(x) => f(x) case None => None })
            match { case Some(y) => g(y) case None => None }

  ==   opt match
         case Some(x) =>
           f(x) match { case Some(y) => g(y) case None => None }
         case None =>
           None match { case Some(y) => g(y) case None => None }
```

Checking the Associative Law (2)

```
==    opt match
        case Some(x) =>
          f(x) match { case Some(y) => g(y) case None => None }
        case None => None
```

Checking the Associative Law (2)

```
==    opt match
        case Some(x) =>
          f(x) match { case Some(y) => g(y) case None => None }
        case None => None

==    opt match
        case Some(x) => f(x).flatMap(g)
        case None => None
```

Checking the Associative Law (2)

```
==    opt match
        case Some(x) =>
          f(x) match { case Some(y) => g(y) case None => None }
        case None => None

==    opt match
        case Some(x) => f(x).flatMap(g)
        case None => None

==    opt.flatMap(x => f(x).flatMap(g))
```

Checking the Associative Law (2)

```
==    opt match
        case Some(x) =>
          f(x) match { case Some(y) => g(y) case None => None }
        case None => None

==    opt match
        case Some(x) => f(x).flatMap(g)
        case None => None

==    opt.flatMap(x => f(x).flatMap(g))

==    opt.flatMap(f(_).flatMap(g))
```

Significance of the Laws for For-Expressions

We have seen that monad-typed expressions are typically written as `for` expressions.

What is the significance of the laws with respect to this?

1. Associativity says essentially that one can "inline" nested for expressions:

```
      for
        y <- for x <- m; y <- f(x) yield y
        z <- g(y)
      yield z

  ==  for x <- m; y <- f(x); z <- g(y)
      yield z
```

Significance of the Laws for For-Expressions

2. Right unit says:

```
      for x <- m yield x

==    m
```

3. Left unit says (more or less):

```
      for y <- unit(x); r <- f(y) yield r

==    f(x)
```

Exceptional Monads

Exceptions

Exceptions in Scala are defined similarly as in Java.

An exception class is any subclass of `java.lang.Throwable`, which has itself subclasses `java.lang.Exception` and `java.lang.Error`. Values of exception classes can be thrown.

```
class BadInput(msg: String) extends Exception(msg)

throw BadInput("missing data")
```

A thrown exception terminates computation, if it is not handled with a `try/catch`.

Handling Exceptions with `try/catch`

A `try/catch` expression consists of a *body* and one or more *handlers*.
Example:

```
def validatedInput(): String =
  try getInput()
  catch
    case BadInput(msg) => println(msg); validatedInput()
    case ex: Exception => println("fatal error; aborting"); throw ex
```

`try/catch` Expressions

An exception is caught by the closest enclosing `catch` handler that matches its type.

This can be formalized with a variant of the substitution model.

Roughly, assuming `ex: Exc`:

```
      try e[throw ex] catch case x: Exc => handler
  -->
      [x := ex]handler
```

Here, `e` is some arbitrary "*evaluation context*" where

- `X` is the next instruction to evaluate in `e[X]`
- `e[X]` does not enclose `X` in a handler that matches `ex`.

Critique of `try/catch`

Exceptions are a low-overhead way for handling abnormal conditions.

But there have also some shortcomings.

- They don't show up in the types of functions that throw them. (in Scala, in Java they do show up in `throws` clauses but that has its own set of downsides).
- They don't work well in parallel computations where we want to communicate an exception from one thread to another.

So in some situations it makes sense to see an exception as a normal function result value, instead of something special.

This idea is implemented in the `scala.util.Try` type.

Handling Exceptions with the `Try` Type

`Try` resembles `Option`, but instead of `Some/None` there is a `Success` case with a value and a `Failure` case that contains an exception:

```
abstract class Try[+T]
case class Success[+T](x: T)       extends Try[T]
case class Failure(ex: Exception) extends Try[Nothing]
```

A primary use of `Try` is as a means of passing between threads and processes the results of computations that can fail with an exception.

Creating a Try

You can wrap up an arbitrary computation in a `Try`.

```
Try(expr)      // gives Success(someValue) or Failure(someException)
```

Here's an implementation of `Try.apply`:

```
import scala.util.control.NonFatal

object Try:
  def apply[T](expr: => T): Try[T] =
    try Success(expr)
    catch case NonFatal(ex) => Failure(ex)
```

Creating a `Try`

You can wrap up an arbitrary computation in a `Try`.

```
Try(expr)     // gives Success(someValue) or Failure(someException)
```

Here's an implementation of `Try.apply`:

```
import scala.util.control.NonFatal

object Try:
  def apply[T](expr: => T): Try[T] =
    try Success(expr)
    catch case NonFatal(ex) => Failure(ex)
```

Here, `NonFatal` matches all exceptions that allow to continue the program.

Composing `Try`

Just like with `Option`, `Try`-valued computations can be composed in for-expressions.

```
for
  x <- computeX
  y <- computeY
yield f(x, y)
```

If `computeX` and `computeY` *both* succeed with results `Success(x)` and `Success(y)`, this returns `Success(f(x, y))`.

If *either* computation fails with an exception `ex`, this returns `Failure(ex)`.

Definition of `flatMap` and `map` on `Try`

```scala
extension [T](xt: Try[T])
  def flatMap[U](f: T => Try[U]): Try[U] = xt match
    case Success(x) => try f(x) catch case NonFatal(ex) => Failure(ex)
    case fail: Failure => fail

  def map[U](f: T => U): Try[U] = xt match
    case Success(x) => Try(f(x))
    case fail: Failure => fail
```

So, for a `Try` value `t`,

```scala
t.map(f)  ==  t.flatMap(x => Try(f(x)))
          ==  t.flatMap(f andThen Try.apply)
```

Exercise

It looks like `Try` might be a monad, with `unit` $=$ `Try.apply`.

Is it?

```
O      Yes
O      No, the associative law fails
O      No, the left unit law fails
O      No, the right unit law fails
O      No, two or more monad laws fail.
```

Exercise

It looks like `Try` might be a monad, with `unit` $=$ `Try.apply`.

Is it?

```
O     Yes
O     No, the associative law fails
X     No, the left unit law fails
O     No, the right unit law fails
O     No, two or more monad laws fail.
```

Solution

It turns out the left unit law fails.

```
Try(expr).flatMap(f)  =?=  f(expr)
```

Indeed the left-hand side will never throw a non-fatal exception whereas the right-hand side will throw any exception thrown by `expr` or `f`.

Hence, `Try` trades one monad law for another law which is more useful in this context:

> *An expression composed from ‘Try‘, ‘map‘, ‘flatMap‘ will never throw a non-fatal exception.*

Call this the “bullet-proof” principle.

Conclusion

We have seen that for-expressions are useful not only for collections.

Many other types also define `map`,`flatMap`, and `withFilter` operations and with them for-expressions.

Examples: `Generator`, `Option`, `Try`.

Many of the types defining `flatMap` are monads.

(If they also define `withFilter`, they are called “monads with zero”).

The three monad laws give useful guidance in the design of library APIs.

Structural Induction on Trees

Structural Induction on Trees

Structural induction is not limited to lists; it applies to any tree structure.

The general induction principle is the following:

To prove a property $P(t)$ for all trees t of a certain type,

- show that $P(l)$ holds for all leaves l of a tree,
- for each type of internal node t with subtrees $s_1, \ldots, s_n$, show that $P(s_1) \land \ldots \land P(s_n)$ *implies* $P(t)$.

Example: IntSets

Recall our definition of `IntSet` with the operations `contains` and `incl`:

```
abstract class IntSet:
  def incl(x: Int): IntSet
  def contains(x: Int): Boolean

object Empty extends IntSet:
  def contains(x: Int): Boolean = false
  def incl(x: Int): IntSet = NonEmpty(x, Empty, Empty)
```

Example: IntSets (2)

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

  def contains(x: Int): Boolean =
    if x < elem then left.contains(x)
    else if x > elem then right.contains(x)
    else true

  def incl(x: Int): IntSet =
    if x < elem then NonEmpty(elem, left.incl(x), right)
    else if x > elem then NonEmpty(elem, left, right.incl(x))
    else this
```

The Laws of IntSet

What does it mean to prove the correctness of this implementation?

One way to define and show the correctness of an implementation consists of proving the laws that it respects.

In the case of `IntSet`, we have the following three laws:

For any set `s`, and elements `x` and `y`:

```
Empty.contains(x)          =  false
s.incl(x).contains(x)      =  true
s.incl(x).contains(y)      =  s.contains(y)        if x != y
```

(In fact, we can show that these laws completely characterize the desired data type).

Proving the Laws of IntSet (1)

How can we prove these laws?

Proposition 1: `Empty.contains(x) = false`.

Proof: According to the definition of `contains` in `Empty`.

Proving the Laws of IntSet (2)

Proposition 2: `s.incl(x).contains(x) = true`

Proof by structural induction on `s`.

Base case: `Empty`

```
Empty.incl(x).contains(x)
```

Proving the Laws of IntSet (2)

Proposition 2: `s.incl(x).contains(x) = true`

Proof by structural induction on `s`.

Base case: `Empty`

```
Empty.incl(x).contains(x)

=   NonEmpty(x, Empty, Empty).contains(x) // by definition of Empty.incl
```

Proving the Laws of IntSet (2)

Proposition 2: `s.incl(x).contains(x) = true`

Proof by structural induction on `s`.

Base case: `Empty`

```
Empty.incl(x).contains(x)

=   NonEmpty(x, Empty, Empty).contains(x) // by definition of Empty.incl

=   true                                  // by definition of NonEmpty.contains
```

Proving the Laws of IntSet (3)

$\boxed{\textbf{Induction step: } \texttt{NonEmpty(x, l, r)}}$

```
NonEmpty(x, l, r).incl(x).contains(x)
```

Proving the Laws of IntSet (3)

Induction step: `NonEmpty(x, l, r)`

```
NonEmpty(x, l, r).incl(x).contains(x)

=   NonEmpty(x, l, r).contains(x)          //  by definition of NonEmpty.incl
```

Proving the Laws of IntSet (3)

Induction step: `NonEmpty(x, l, r)`

```
NonEmpty(x, l, r).incl(x).contains(x)

=   NonEmpty(x, l, r).contains(x)          //  by definition of NonEmpty.incl

=   true                                   //  by definition of NonEmpty.contains
```

Proving the Laws of IntSet (4)

Induction step: `NonEmpty(y, l, r)` **where** `y < x`

```
NonEmpty(y, l, r).incl(x).contains(x)
```

Proving the Laws of IntSet (4)

Induction step: `NonEmpty(y, l, r)` **where** `y < x`

```
NonEmpty(y, l, r).incl(x).contains(x)

=   NonEmpty(y, l, r.incl(x)).contains(x) // by definition of NonEmpty.incl
```

Proving the Laws of IntSet (4)

Induction step: `NonEmpty(y, l, r)` **where** `y < x`

```
NonEmpty(y, l, r).incl(x).contains(x)

=   NonEmpty(y, l, r.incl(x)).contains(x) // by definition of NonEmpty.incl

=   r.incl(x).contains(x)                 // by definition of NonEmpty.contains
```

Proving the Laws of IntSet (4)

Induction step: `NonEmpty(y, l, r)` **where** `y < x`

```
NonEmpty(y, l, r).incl(x).contains(x)

=   NonEmpty(y, l, r.incl(x)).contains(x) // by definition of NonEmpty.incl

=   r.incl(x).contains(x)                 // by definition of NonEmpty.contains

=   true                                  // by the induction hypothesis
```

Proving the Laws of IntSet (4)

Induction step: `NonEmpty(y, l, r)` **where** `y < x`

```
NonEmpty(y, l, r).incl(x).contains(x)

=   NonEmpty(y, l, r.incl(x)).contains(x) // by definition of NonEmpty.incl

=   r.incl(x).contains(x)                 // by definition of NonEmpty.contains

=   true                                  // by the induction hypothesis
```

Induction step: `NonEmpty(y, l, r)` **where** `y > x` is analogous

Proving the Laws of IntSet (5)

Proposition 3: If `x != y` then

```
xs.incl(y).contains(x)  =  xs.contains(x).
```

Proof by structural induction on `s`. Assume that `y < x` (the dual case `x < y` is analogous).

Base case: `Empty`

```
Empty.incl(y).contains(x)                  // to show: =  Empty.contains(x)
```

Proving the Laws of IntSet (5)

Proposition 3: If `x != y` then

```
xs.incl(y).contains(x)  =  xs.contains(x).
```

Proof by structural induction on `s`. Assume that `y < x` (the dual case `x < y` is analogous).

Base case: `Empty`

```
Empty.incl(y).contains(x)                    // to show: =  Empty.contains(x)

=   NonEmpty(y, Empty, Empty).contains(x)  // by definition of Empty.incl
```

Proving the Laws of IntSet (5)

Proposition 3: If `x != y` then

```
xs.incl(y).contains(x)  =  xs.contains(x).
```

Proof by structural induction on `s`. Assume that `y < x` (the dual case `x < y` is analogous).

$\boxed{\textbf{Base case: } \texttt{Empty}}$

```
Empty.incl(y).contains(x)                   // to show: =  Empty.contains(x)

=   NonEmpty(y, Empty, Empty).contains(x)   // by definition of Empty.incl

=   Empty.contains(x)                       // by definition of NonEmpty.contains
```

Proving the Laws of IntSet (6)

For the inductive step, we need to consider a tree `NonEmpty(z, l, r)`. We distinguish five cases:

1. `z = x`
2. `z = y`
3. `z < y < x`
4. `y < z < x`
5. `y < x < z`

First Two Cases: `z = x` then `z = y`

Induction step: `NonEmpty(x, l, r)`

```
NonEmpty(x, l, r).incl(y).contains(x)  // to show: = NonEmpty(x,l,r).contains(x)
```

First Two Cases: `z = x` then `z = y`

Induction step: `NonEmpty(x, l, r)`

```
NonEmpty(x, l, r).incl(y).contains(x)  // to show: = NonEmpty(x,l,r).contains(x)

=   NonEmpty(x, l.incl(y), r).contains(x) // by definition of NonEmpty.incl
```

First Two Cases: `z = x` then `z = y`

Induction step: `NonEmpty(x, l, r)`

```
NonEmpty(x, l, r).incl(y).contains(x)  // to show: = NonEmpty(x,l,r).contains(x)

=   NonEmpty(x, l.incl(y), r).contains(x) // by definition of NonEmpty.incl

=   true                                  // by definition of NonEmpty.contains
```

First Two Cases: `z = x` then `z = y`

Induction step: `NonEmpty(x, l, r)`

```
NonEmpty(x, l, r).incl(y).contains(x)  // to show: = NonEmpty(x,l,r).contains(x)

=   NonEmpty(x, l.incl(y), r).contains(x) // by definition of NonEmpty.incl

=   true                                  // by definition of NonEmpty.contains

=   NonEmpty(x, l, r).contains(x)         // by definition of NonEmpty.contains
```

First Two Cases: `z = x` then `z = y`

Induction step: `NonEmpty(x, l, r)`

```
NonEmpty(x, l, r).incl(y).contains(x)  // to show: = NonEmpty(x,l,r).contains(x)

=   NonEmpty(x, l.incl(y), r).contains(x) // by definition of NonEmpty.incl

=   true                                  // by definition of NonEmpty.contains

=   NonEmpty(x, l, r).contains(x)         // by definition of NonEmpty.contains
```

Induction step: `NonEmpty(y, l, r)`

```
NonEmpty(y, l, r).incl(y).contains(x)  // to show: = NonEmpty(y,l,r).contains(x)
```

First Two Cases: `z = x` then `z = y`

Induction step: `NonEmpty(x, l, r)`

```
NonEmpty(x, l, r).incl(y).contains(x)  // to show: = NonEmpty(x,l,r).contains(x)

=   NonEmpty(x, l.incl(y), r).contains(x) // by definition of NonEmpty.incl

=   true                                  // by definition of NonEmpty.contains

=   NonEmpty(x, l, r).contains(x)         // by definition of NonEmpty.contains
```

Induction step: `NonEmpty(y, l, r)`

```
NonEmpty(y, l, r).incl(y).contains(x)  // to show: = NonEmpty(y,l,r).contains(x)

=    NonEmpty(y, l, r).contains(x)        // by definition of NonEmpty.incl
```

Case z < y

Induction step: `NonEmpty(z, l, r)` **where** `z < y < x`

```
NonEmpty(z, l, r).incl(y).contains(x)  // to show: = NonEmpty(z,l,r).contains(x)
```

Case z < y

Induction step: `NonEmpty(z, l, r)` **where** `z < y < x`

```
NonEmpty(z, l, r).incl(y).contains(x)  // to show: = NonEmpty(z,l,r).contains(x)

=   NonEmpty(z, l, r.incl(y)).contains(x)  // by definition of NonEmpty.incl
```

Case z < y

Induction step: `NonEmpty(z, l, r)` **where** `z < y < x`

```
NonEmpty(z, l, r).incl(y).contains(x)  // to show: = NonEmpty(z,l,r).contains(x)

=   NonEmpty(z, l, r.incl(y)).contains(x)  // by definition of NonEmpty.incl

=   r.incl(y).contains(x)                  // by definition of NonEmpty.contains
```

Case z < y

Induction step: `NonEmpty(z, l, r)` **where** `z < y < x`

```
NonEmpty(z, l, r).incl(y).contains(x)  // to show: = NonEmpty(z,l,r).contains(x)

=   NonEmpty(z, l, r.incl(y)).contains(x)  // by definition of NonEmpty.incl

=   r.incl(y).contains(x)                  // by definition of NonEmpty.contains

=   r.contains(x)                          // by the induction hypothesis
```

Case z < y

Induction step: `NonEmpty(z, l, r)` **where** `z < y < x`

```
NonEmpty(z, l, r).incl(y).contains(x)  // to show: = NonEmpty(z,l,r).contains(x)

=   NonEmpty(z, l, r.incl(y)).contains(x)  // by definition of NonEmpty.incl

=   r.incl(y).contains(x)                  // by definition of NonEmpty.contains

=   r.contains(x)                          // by the induction hypothesis

=   NonEmpty(z, l, r).contains(x)          // by definition of NonEmpty.contains
```

Case y < z < x

Induction step: `NonEmpty(z, l, r)` **where** `y < z < x`

```
NonEmpty(z, l, r).incl(y).contains(x)   // to show: = NonEmpty(z,l,r).contains(x)
```

Case y < z < x

Induction step: `NonEmpty(z, l, r)` **where** `y < z < x`

```
NonEmpty(z, l, r).incl(y).contains(x)   // to show: = NonEmpty(z,l,r).contains(x)

=   NonEmpty(z, l.incl(y), r).contains(x)  // by definition of NonEmpty.incl
```

Case `y < z < x`

Induction step: `NonEmpty(z, l, r)` **where** `y < z < x`

```
NonEmpty(z, l, r).incl(y).contains(x)   // to show: = NonEmpty(z,l,r).contains(x)

=   NonEmpty(z, l.incl(y), r).contains(x)  // by definition of NonEmpty.incl

=   r.contains(x)                          // by definition of NonEmpty.contains
```

Case y < z < x

Induction step: NonEmpty(z, l, r) **where** y < z < x

```
NonEmpty(z, l, r).incl(y).contains(x)   // to show: = NonEmpty(z,l,r).contains(x)

=   NonEmpty(z, l.incl(y), r).contains(x)  // by definition of NonEmpty.incl

=   r.contains(x)                          // by definition of NonEmpty.contains

=   NonEmpty(z, l, r).contains(x)          // by definition of NonEmpty.contains
```

Case x < z

Induction step: `NonEmpty(z, l, r)` **where** `y < x < z`

```
NonEmpty(z, l, r).incl(y).contains(x)  // to show: = NonEmpty(z,l,r).contains(x)
```

Case x < z

Induction step: `NonEmpty(z, l, r)` **where** `y < x < z`

```
NonEmpty(z, l, r).incl(y).contains(x)  // to show: = NonEmpty(z,l,r).contains(x)

=   NonEmpty(z, l.incl(y), r).contains(x)  // by definition of NonEmpty.incl
```

Case x < z

Induction step: `NonEmpty(z, l, r)` **where** `y < x < z`

```
NonEmpty(z, l, r).incl(y).contains(x)  // to show: = NonEmpty(z,l,r).contains(x)

=   NonEmpty(z, l.incl(y), r).contains(x)  // by definition of NonEmpty.incl

=   l.incl(y).contains(x)                  // by definition of NonEmpty.contains
```

Case x < z

Induction step: `NonEmpty(z, l, r)` **where** `y < x < z`

```
NonEmpty(z, l, r).incl(y).contains(x)  // to show: = NonEmpty(z,l,r).contains(x)

=   NonEmpty(z, l.incl(y), r).contains(x)  // by definition of NonEmpty.incl

=   l.incl(y).contains(x)                  // by definition of NonEmpty.contains

=   l.contains(x)                          // by the induction hypothesis
```

Case x < z

Induction step: `NonEmpty(z, l, r)` **where** `y < x < z`

```
NonEmpty(z, l, r).incl(y).contains(x)  // to show: = NonEmpty(z,l,r).contains(x)

=   NonEmpty(z, l.incl(y), r).contains(x)  // by definition of NonEmpty.incl

=   l.incl(y).contains(x)                  // by definition of NonEmpty.contains

=   l.contains(x)                          // by the induction hypothesis

=   NonEmpty(z, l, r).contains(x)          // by definition of NonEmpty.contains
```

These are all the cases, so the proposition is established.

Exercise (Hard)

Suppose we add a function `union` to `IntSet`:

```
abstract class IntSet:
  ...
  def union(other: IntSet): IntSet

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

class NonEmpty(x: Int, l: IntSet, r: IntSet) extends IntSet:
  ...
  def union(other: IntSet): IntSet = l.union(r.union(other)).incl(x)
```

Exercise (Hard)

The correctness of `union` can be translated into the following law:

Proposition 4:

```
xs.union(ys).contains(x)  =  xs.contains(x) || ys.contains(x)
```

Show proposition 4 by using structural induction on `xs`.