

# Intuitionistic Logic Implemented in Scala

## Submitted for Publication at SAI Computing Conference, London, 2021

Vlad Patryshev

### ABSTRACT

*An implementation of intuitionistic (Grothendieck topos-based, for a finite site) is described, with a variety of examples.*

*This paper demonstrates how non-Boolean logic can be implemented, to enable using all the power of Intuitionism. For that, a library was developed that works with topos logic. Sample code shows the logic for a variety of sites.*

## 1 Introduction

In this paper I present a Scala implementation of topos logic (that is, an intuitionistic logic). Intuitionism includes all kinds of fuzzy and temporal logic (see TLA+). Having a code package modeling Intuitionism, one can experiment with different kinds of fuzziness and different shapes of time flow. The toposes used are Grothendieck toposes, and the sites are finite.

Two categories (toposes) are chosen over which Grothendieck toposes are built. These categories are *Set* and *Set<sub>f</sub>*, that is, a category of all (available in the code) sets, and the category of all finite sets available in the code. Since Scala does not exactly implement sets as are defined in set theories, a couple of new classes are introduced to resolve this issue.

This paper demonstrates how non-Boolean logic can be implemented, to enable using all the power of Intuitionism. For that, a library was developed that works with topos logic. Sample code shows the logic for a variety of sites. The code is located at <https://github.com/vpatryshev/Categories>.

## 2 Non-Boolean Logic

Let us first introduce the idea of non-Booleanness, in a strict mathematical sense of the term.

### 2.1 The Meaning of Non-Booleanness

Traditionally, as soon as we switch from the complicated logic of the real world into the realm of formal software design, we assume that just two outcomes are possible for any statement: it is either true or false. This is not a feature of mathematical discourse but rather an interesting feature of the modern way of thinking. In real life, we admit that there are more possibilities: something we may not know today, we may learn tomorrow. We may assume something to be true with a certain probability. Similarly, in science we may not know the answer, and the answer may be something we don't expect - like with the case of Continuum Hypothesis.

In intuitionistic logic, the double negation law does not have to hold. If it does, the logic is Boolean. Here we will show a way to eliminate the double negation law while keeping the remaining rules and axioms that don't depend on double negation. Of course, if the double negation law is included, we still have an intuitionistic logic that is also Boolean.

Also a logic may have more than two logical constants, but that does not make it non-Boolean. The double negation rule may still be applicable in such a logic.

#### 2.1.1 Example 1. Boolean Logic, but not 2-Valued

Consider bytes and their operations: bitwise conjunction, bitwise disjunction, and negation. There are 256 different values, but we know that double negation in this bitwise logic is an identity. So, this 256-valued logic is Boolean.

## 2.2 Dropping Booleanness

Remove the rule stating that either  $P$  or  $\neg P$  is true. Formally, that  $\vdash P \vee \neg P$ . Once we remove this rule, we cannot apply it anymore in general settings, although it may still work in some cases.

We immediately bump into a problem: this rule was used to define implication:  $P \rightarrow Q \equiv \neg P \vee Q$ . We must define implication differently. Here's a reasonable alternative solution:

$$(P \wedge Q) \vdash R \equiv P \vdash (Q \rightarrow R)$$

Informally, saying that  $R$  can be deduced from a conjunction  $P \wedge Q$  is the same as saying that we can deduce the implication  $Q \rightarrow R$  from  $P$ . Imagine  $Q$  is  $\top$ . Then, from the formula above, it follows that  $Q \rightarrow R$  is the same as  $R$ . On the other hand, if  $Q$  is closer to the bottom, the situation may change. For example, if  $Q$  is below  $R$ , the implication  $Q \rightarrow R$  is true. We can interpret the implication defined this way as the level of dependency of  $R$  on  $Q$ .

As you see, we can define implication via conjunction, and negation is not involved.

Does this remind you currying? Given a function  $f(P, Q) : R$ , we transform it to a function  $f_c(P) : Q \rightarrow R$ .

Now that implication is defined, we can define negation as  $\neg P = P \rightarrow \perp$ . Its properties make it similar to classical negation, but the double negation law is not generally applicable anymore. Assume we use the definition above. In the table below, we list some properties that are valid for this definition.

Statement	Meaning
$P \wedge \neg P \vdash \perp$	Negation of $P$ is incompatible with $P$ .
$\neg \neg \neg P \vdash \neg P$	Triple negation is the same as single negation.
$P \vdash \neg \neg P$	Double negation of $P$ is weaker than $P$ .
$\neg P \vee \neg Q \vdash \neg(P \wedge Q)$	If not $P$ or not $Q$ , then we cant have both $P$ and $Q$
$\neg(P \vee Q) \vdash \neg P \wedge \neg Q$	If disjunction of $P$ and $Q$ is not true, then neither $P$ nor $Q$ is true.
$\neg P \wedge \neg Q \vdash \neg(P \vee Q)$	If neither $P$ nor $Q$ is true, their disjunction is not true.

Some properties do not apply in this kind of logic:

Wrong Statement	Whats wrong with it?
$\neg \neg P \vdash P$	Negation of negation of $P$ is not strong enough to give us $P$ .
$\neg(P \wedge Q) \vdash \neg P \vee \neg Q$	Even if we cant have both at the same time, this does not mean that one of them is always wrong.

### 2.2.1 Example 2. Three Logical Values (Ternary Logic)

Start with three logical values,  $\top$ ,  $\perp$ , and  $?$ . Define operations for them:

$x$	$\neg x$	$\neg \neg x$
$\top$	$\perp$	$\top$
$?$	$\perp$	$\top$
$\perp$	$\top$	$\perp$

<b>x</b>	<b>y</b>	<b>x ∧ y</b>	<b>x ∨ y</b>	<b>x → y</b>
⊤	⊤	⊤	⊤	⊤
⊤	?	?	⊤	?
⊤	⊥	⊥	⊤	⊥
?	⊤	?	⊤	⊤
?	?	?	?	⊤
?	⊥	⊥	?	⊥
⊥	⊤	⊥	⊤	⊤
⊥	?	⊥	?	⊤
⊥	⊥	⊥	⊥	⊤

This is the simplest intuitionistic logic, and it is a good tool for testing our statements. The value ?, or unknown, is somewhere between truth ⊤ and false ⊥.

### 2.2.2 Example 3. Infinite Number of Logical Values

Take  $[0, 1]$ , the set of all real numbers between 0 and 1, including 0 and 1. We can turn it into logic by defining:

- $\perp = 0$
- $\top = 1$
- $a \wedge b = glb(a, b)$
- $a \vee b = lub(a, b)$

In this example, implication is defined, as always, via the equivalence:  $(x \wedge y) \leq z \equiv x \leq (y \rightarrow z)$ .

We consider the following two cases separately:

First, when  $y \leq z$ ,  $(x \wedge y) \leq z$  is always true, for any  $x$ . Hence  $y \rightarrow z$  cannot be smaller than any  $x$ , that is, it must be the top element, 1.

In the opposite case, when  $y$  is bigger than  $z$  (we are talking about numbers),  $(x \wedge y \leq z) \equiv (x \leq z)$ , so now we will have  $x \leq z \equiv x \leq (y \rightarrow z)$ , which means that  $y \rightarrow z = z$ .

Remember that we defined negation,  $\neg x$ , as  $x \rightarrow 0$ , and we have  $\neg x \equiv \text{if } (x = 0) \text{ 1 else } 0$ . You can also check whether double negation maps 0 to 0 and any non-zero value to 1.

In this example, we started with a set of values that was popular in "fuzzy logic," and we were able to build a pretty sound logic, except that it could not be made Boolean: double negation is not an identity.

## 3 Sets and Categories

### 3.1 Sets

Scala programming language uses a pretty loose notion of sets; they have too many methods (like `head()`), and functions are not defined as "special sets of pairs", as they are defined in ZFC. So we had to introduce a special class for working with sets. Class *BigSet* is the most general class defining sets; it does not even have to be enumerable.

```
1 abstract class BigSet[T] extends Set[T] {
2   override def size: Int = Sets.InfiniteSize
3   ...
}
```

This abstract "container" covers all possible sets. Remember, sets are not typed (in a set theory).

```
1 object Sets {
2   type set = Set[Any]
3
4   implicit class untype[T](s: Set[T]) {
5     def untyped: set = s.asInstanceOf[set]
6   }
7
8   def isFinite(s: Set[_]) Boolean = s.size != InfiniteSize
9   val Empty: set = Set.empty[Any]
10  val Unit: set = Set(Empty)
11  val FiniteSets = BigSet.comprehension(isFinite)
12  ...
}
```

We can now define non-enumerable sets (thus banning all those Scala/Java methods that enumerate):

```
1 trait NonEnumerableSet[T] extends Set[T] {
2   private def notEnumerable = throw Stone(...)
3   override def isEmpty: Boolean = notEnumerable
4   def iterator: Iterator[T] = notEnumerable
5   override def toArray[S >: T : ClassTag] = notEnumerable
}
```

```
6 }
```

These classes will be used to properly define Grothendieck toposes over sites.

## 4 Categories

Before defining Category, let's define Graph:

```
1 trait Graph { graph =>
2   type Node
3   type Arrow
4   def nodes: Set[Node]
5   def arrows: Set[Arrow]
6   def d0(f: Arrow): Node
7   def d1(f: Arrow): Node
8   ...
```

trait Graph implements a pretty big number of useful methods, but what's important is that it specifies Nodes, Arrows, and their relationships.

Having a type of Graphs, we can define a Category in Scala like this:

```
1 abstract class Category(override val name: String) extends Graph {
2   type Obj = Node
3   def id(o: Obj): Arrow
4   def m(f: Arrow, g: Arrow): Option[Arrow] // composition
5
6   lazy val op: Category = {
7     val src = this
8     new Category(s"$name") {
9       override def id(o: Obj): Arrow = src.id(o)
10      override def m(f: Arrow, g: Arrow) = src.m(f, g)
11    }
12 }
```

Note that composition of arrows is a partial function lifted to Option.

A category does not necessarily have to be finite: here is an example of an infinite one:

```
1 object N extends BigSet[BigInt] with EnumerableSet[BigInt] {
2   override def iterator: Iterator[BigInt] = new Iterator[BigInt] {...}
3   override def contains(n: BigInt): Boolean = n >= 0
4 }
5 object PoSet {
6   lazy val ofNaturalNumbers: PoSet[BigInt] = new PoSet(N, comparator) {...}
7 }
8 lazy val NaturalNumbers: Category = fromPoset("N", PoSet.ofNaturalNumbers)
```

We should never attempt to materialize such a category.

That was a countable category; we can even build an uncountable one:

```
1 class SetCategory(objects: BigSet[Set[Any]])
2   extends Category(name="Sets", graphOfSets(objects)) {
3   type Node = set
4   type Arrow = SetFunction
5
6   override def d0(f: SetFunction): set = f.d0
7   override def d1(f: SetFunction): set = f.d1
8   override def m(f: Arrow, g: Arrow): Option[Arrow] = f.compose g
9   override def id(s: set): SetFunction = SetFunction.id(s)
10  override def toString: String = "Category of All Sets"
11  ...
12 object Setf extends SetCategory(FiniteSets) {
13   override def toString: String = "Category of Finite Sets"
14 }
```

But what is SetFunction?

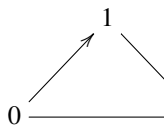
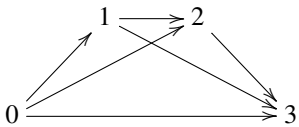
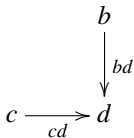
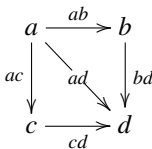
```

1 case class SetFunction(
2   override val tag: String,
3   override val d0: set,
4   override val d1: set,
5   mapping: Any => Any) extends SetMorphism { self =>
6   ...
7 def compose(g: SetFunction): Option[SetFunction] =
8   if (d1 == g.d0) {
9     Some(new SetFunction(newTag, d0, g.d1, (x: Any) => g(self(x))))
10  } else None

```

## 4.1 Samples of Categories

Below are some example of categories, built using these tools:

0		<code>val _0_ : Cat = segment(0)</code>
1		<code>val _1_ : Cat = segment(1)</code>
1 + 1		<code>val _1plus1_ : Cat = discrete(Set("a", "b"))</code>
2	$0 \longrightarrow 1$	<code>val _2_ : Cat = segment(2)</code>
3		<code>val _3_ : Cat = segment(3)</code>
4		<code>val _4_ : Cat = segment(4)</code>
<i>ParallelPair</i>	$0 \begin{matrix} \xrightarrow{a} \\ \xrightarrow{b} \end{matrix} 1$	<code>category"ParallelPair:({0,1}, {a:0-&gt;1, b:0-&gt;1})"</code>
<i>Pullback</i>		<code>category"Pullback:({b,c,d}, {bd:b-&gt;d, cd:c-&gt;d})"</code>
<i>Square</i>		<code>category"Square:({a,b,c,d}, {ab:a-&gt;b, ac:a-&gt;c, bd:b-&gt;d, cd:c-&gt;d, ad:a-&gt;d}, {bd o ab = ad, cd o ac = ad})"</code>

## 4.2 Functors and Natural Transformations

Definitions are trivial:

```

1 abstract class Functor(
2   val d0: Category, val d1: Category
3 ) extends Morphism {
4   def objectsMapping(x: d0.Obj): d1.Obj
5   def arrowsMapping(a: d0.Arrow): d1.Arrow
6   def compose(next: Functor): Option[Functor] = {
7     if (this.d1 != next.d0) None else Some { //...

```

```

1 abstract class NaturalTransformation
2   extends Morphism[Functor, Functor] {

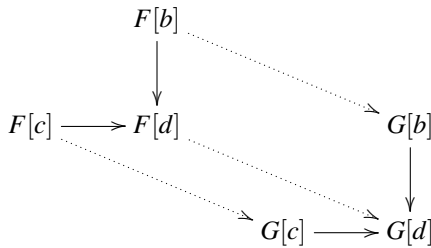
```

```

3
4
5 val domain: Category = d0.d0
6 val codomain: Category = d1.d1 // == d0.d1
7
8 def transformPerObject(x: domain.Obj): codomain.Arrow

```

In the picture, you see a natural transformation for category "Pullback":



## 5 Copresheaves (aka Diagrams)

### 5.1 Definition

A *Copresheaf*, also known as *Diagram*, of category **C** in category **D** is a functor from **C** to **D**. We will, further on, be dealing with copresheaves in **Set**, and call them *diagrams*.

```

1 abstract class Diagram(val domain: Category)
2   extends Functor(tag, topos.domain, Setf) { diagram =>
3
4   def isElementOf(other: Diagram): Boolean =
5     d0.objects.forall { o => other(o) contains (this(o)) }
6
7   def isContainedIn(other: Diagram): Boolean =
8     d0.objects.forall { o => this(o) subsetOf other(o) }

```

### 5.2 Example 5, Diagrams over $\mathbb{N}$

A functor  $\mathbb{N} \rightarrow \mathbf{Sets}$  consists of sets  $F[i]$  and functions  $F[i] \rightarrow F[i+1]$ . (Other functions are compositions of these). These diagrams model discrete time flow : discrete,  $[1]$ . Thats the basic type of TLA+ book.

$$F[0] \longrightarrow F[1] \longrightarrow F[2] \longrightarrow \dots$$

Below is a table of some shapes of time that we encounter in practice.

$F[0] \longrightarrow F[1] \longrightarrow F[2] \longrightarrow \dots$	TLA+
$\bullet$	Just sets
$0 \longrightarrow 1$	Single transition (today and tomorrow)
$\begin{array}{ccc} & b & \\ & \downarrow bd & \\ c & \longrightarrow & d \\ & cd & \end{array}$	Git Merge
$\begin{array}{ccc} a & \xrightarrow{ab} & b \\ & \searrow ad & \downarrow bd \\ ac & \downarrow & d \\ c & \xrightarrow{cd} & d \end{array}$	Eventual Consistency
$\begin{array}{ccccc} & & 6:00 & & \\ & & \curvearrowright & & \\ Feb1 & \longrightarrow & Feb2 & \longrightarrow & Feb3 \end{array}$	Groundhog day

## 6 Logic in Grothendieck Topos

Now that we have a category of diagrams (it's a Grothendieck topos), we can dive deeper and build logic in it. We will be using points of objects to illustrate logic. Generally speaking, a Grothendieck topos is not well-pointed, though.

### 6.1 Point of a Diagram

A point is any arrow  $1 \longrightarrow D$ , where  $1$  is a terminal object of the category of diagrams.

```
1 class Point(val tag: Any, val mapping: Any => Any)
```

### 6.2 Subobject Classifier

This is a special object in a category of diagrams contains all the logic of a topos; we just have to discover it.

$$\begin{array}{ccc} A & \longrightarrow & 1 \\ f \downarrow & & \downarrow \text{true} \\ B & \xrightarrow{\chi_f} & \Omega \end{array}$$

How can we build it?

$\Omega(x) \equiv \text{hom}(hx, \Omega)$  - by Yoneda lemma

$\text{hom}(hx, \Omega) \equiv p \subset hx$  - by definition of  $\Omega$ , see above.

```
1 case class Representable(x: domain.Obj)
2 extends Diagram(s"hom($x, _)", domain)
3 //...
4
5 object Omega extends Diagram("Omega", this, domain) {
6   val False: Point = points.head named "false"
7   val True: Point = points.last named "true"
8   val conjunction: DiagramArrow = ... // Omega x Omega (x) => Omega (x)
9   val disjunction: DiagramArrow = ... // x (x) => Omega (x)
10  al implication: DiagramArrow = ... // x (x) => Omega (x)
11  ...
12 }
13 val Omega x Omega = product2(Omega, Omega)
14
15 val DiagonalOfOmega: DiagramArrow =
16   buildArrow("Diagonal", Omega, Omega x Omega, _ => (s: Any) => (s, s))
```

#### 6.2.1 How to Build Conjunction in $\Omega$

$$\begin{array}{ccc} 1 & \longrightarrow & 1 \\ (true, true) \downarrow & & \downarrow \text{true} \\ \Omega \times \Omega & \xrightarrow{\wedge} & \Omega \end{array}$$

#### 6.2.2 How to Build Implication in $\Omega$

Start with building a subobject  $\Omega_1$  defining a partial order on  $\Omega$ :

$$\Omega \xrightarrow{p_1} \Omega \times \Omega \xrightarrow{p_1} \Omega$$

Now, implication is the arrow that classifies  $\Omega_1$ :

$$\begin{array}{ccc} \Omega_1 & \longrightarrow & 1 \\ \downarrow & & \downarrow \text{true} \\ \Omega \times \Omega & \xrightarrow{\text{implication}} & \Omega \end{array}$$

```
1 lazy val implication: DiagramArrow = {
2   val inclusion: DiagramArrow = inclusionOf(Omega_1) in Omega x Omega iHope
3
4   classifies(inclusion, ">=")
5 }
```

### 6.3 Predicate Logic

First, what is a predicate in a topos? It's just any arrow  $D \rightarrow \Omega$ .

Logical operations on  $\Omega$  provide operations on predicates:

```
1 trait Predicate extends DiagramArrow { p: DiagramArrow =>
2   val d0: Diagram
3   val d1: Diagram = Omega
4   def and(q: Predicate): Predicate = binaryOp(q, "&", Omega.conjunction)
5   def or(q: Predicate): Predicate = binaryOp(q, "|", Omega.disjunction)
6   def ==>(q: Predicate): Predicate = binaryOp(q, "==>", Omega.implication)
7   ...
8 }
9
10 def not(p: Predicate): Predicate = p ==> FalsePredicate
```

As an example, here's a test case from a unittest for this code:

```
1 def check(c: Category): MatchResult[Any] = {
2   val topos = new CategoryOfDiagrams(c)
3   import topos._
4
5   for { p <- Omega.points map (_.asPredicate) } {
6     (True ==> p)      === p
7     (False ==> p)     === True
8     (p ==> p)         === True
9     (p ==> True)      === True
10    not(not(not(p)))  === not(p)
11
12    for { q <- Omega.points map (_.asPredicate)
13          r <- Omega.points map (_.asPredicate) } {
14      (p and q ==> r) === (p ==> (q ==> r))
15    }
16  }
17 }
```

## 7 Conclusion

Neither Scala nor Java was designed to deal with problems like these: typeless sets, categories, infinite collections. Also, the higher order type system in Scala 2 is not very convenient for the problems like the ones I encountered implementing categories and toposes; dependent types would make more sense. There's a hope that Scala 3 will help dealing with it properly. Strangely, implementing it in Haskell was even more challenging.

In addition to intuitionistic logic, Lawvere topologies can be implemented based on this package; and the implementation is now available in the same repository. Using an older version of this software, we managed to enumerate all Lawvere topologies over finite sets.

## References

- [1] Lamport, L., 2003. *Specifying Systems*. Pearson Education. <https://lamport.azurewebsites.net/tla/book.html>.
- [2] Milewski, B., 2018. *Category Theory for Programmers*. <https://github.com/hmemcpy/milewski-ctfp-pdf/releases>.
- [3] Pierce, B., 1991. *Basic Category Theory for Computer Scientists (Foundations of Computing)*. The MIT Press, Reading, MA.
- [4] Johnstone, P., 1977. *Topos Theory*. Academic Press.
- [5] MacLane, S., and Moerdijk, I., 1992. *Sheaves in Geometry and Logic: A First Introduction to Topos Theory*. Springer.
- [6] Patryshev, V., 2019. *A Brief Course in Modern Math for Programmers*. lcpres. <https://gumroad.com/l/lcbk02>.
- [7] nLab, 2008. Subobject classifier. <https://ncatlab.org/nlab/show/subobjectclassifier+>.
- [8] Patryshev, V., 2020. Categories in scala. <https://github.com/vpatryshev/Categories>.
- [9] Patryshev, V., 1984. "On the grothendieck topologies in the toposes of presheaves". *Cahiers de topologie et gomtrie differentielle catgoriques*, **25**(2), pp. 207–215.