Logic in Sole.jl

In [1]:
```
using Pkg
Pkg.activate("..")
Pkg.instantiate()
Pkg.update()
```

```
 Activating project at `~/logic-and-machine-learning`
   Updating registry at `~/.julia/registries/General.toml`
   Updating git-repo `https://github.com/aclai-lab/ManyExpertDecisionTree
s.jl`
   Updating git-repo `https://github.com/aclai-lab/SoleReasoners.jl#embed
ding`
  No Changes to `~/logic-and-machine-learning/Project.toml`
  No Changes to `~/logic-and-machine-learning/Manifest.toml`
```

SoleLogics.jl

`SoleLogics.jl` is not only the package in the `Sole.jl` framework specifically developed for logic: it is the core library of the framework itself.

In a nutshell, it provides a fresh codebase for computational logic, featuring easy manipulation of:

- Propositional and (multi)modal logics (atoms, logical constants, alphabets, grammars, crisp/fuzzy algebras);
- Logical formulas (parsing, random generation, minimization);
- Logical interpretations (propositional valuations, Kripke structures);
- Algorithms for finite model checking, that is, checking that a formula is satisfied by an interpretation.

In this notebook, we will see examples of all these functionalities, providing a comprehensive overview of the whole package.

In [2]:
```
using SoleLogics
```

Propositional Logic

Formulas and Intepretations

In [3]:
```
p = Atom("it's alive!")
```

Out[3]:
```
Atom{String}: it's alive!
```

In [4]:
```
q = Atom("it's mortal!")
```

Out[4]:
```
Atom{String}: it's mortal!
```

In [5]:
```
φ = p ∧ q
```

```
Out[5]:  SyntaxBranch: it's alive! ∧ it's mortal!
```

```
In [6]:  φ isa Formula && p isa Formula
```

```
Out[6]:  true
```

φ is the root node of a syntax tree.

```
In [7]:  token(φ)    # Print the syntax token at the root node
```

```
Out[7]:  ∧
```

```
In [8]:  children(φ) # Print the children of the root node
```

```
Out[8]:  (Atom{String}: it's alive!, Atom{String}: it's mortal!)
```

Let's create a method for negating any formula.

```
In [9]:  function negateformula(f::Formula)
             return ¬f
         end
```

```
Out[9]:  negateformula (generic function with 1 method)
```

```
In [10]:  negateformula(φ)
```

```
Out[10]:  SyntaxBranch: ¬(it's alive! ∧ it's mortal!)
```

```
In [11]:  negateformula(p)
```

```
Out[11]:  SyntaxBranch: ¬it's alive!
```

We can use `syntaxstring` to obtain the string representation of a `Formula` .

```
In [12]:  syntaxstring(φ)
```

```
Out[12]:  "it's alive! ∧ it's mortal!"
```

We can also parse `Formula` s from standard string representations.

```
In [13]:  φ = parseformula("it's alive! ∧ it's mortal!")
```

```
Out[13]:  SyntaxBranch: it's alive! ∧ it's mortal!
```

Let's see how we can build our own string representation.

Spoiler: it's a good example of Julia's multiple dispatch in action!

```
In [14]:  function my_own_string_representation(f::Atom)
              return syntaxstring(f)
          end

          function my_own_string_representation(::NamedConnective{:¬}, children)
              return "It is not the case that " *
              "$(my_own_string_representation(first(children)))"
          end
```

```
function my_own_string_representation(::NamedConnective{:∧}, children)
    subformula1, subformula2 = children
    return "both $(my_own_string_representation(subformula1)) and " *
    "$(my_own_string_representation(subformula2))"
end

function my_own_string_representation(f::SyntaxBranch)
    return my_own_string_representation(token(f), children(f))
end
```

Out[14]: `my_own_string_representation (generic function with 4 methods)`

In [15]:
```
my_own_string_representation(¬φ)
```

Out[15]: `"It is not the case that both it's alive! and it's mortal!"`

Use use `TruthDict` structures (truth dictionaries, or dictionaries of truths) to associate to each atom il our alphabet a truth value.

You can see these as properties of an object, or features/attributes of an instance!

In [16]:
```
soul = TruthDict([p => true, q => false])
```

Out[16]: `TruthDict with values:`

it's mortal! String	**it's alive!** String
⊥	⊤

In [17]:
```
body = TruthDict([p => true, q => true])
```

Out[17]: `TruthDict with values:`

it's mortal! String	**it's alive!** String
⊤	⊤

But what is `⊤`?

In [18]:
```
⊤ isa Truth       &&      ⊤ isa Formula
```

Out[18]: `true`

If you find these Unicode characters uncomfortable to work with, they have aliases:

In [19]:
```
(TOP, BOT, CONJUNCTION, DISJUNCTION, IMPLICATION)
```

Out[19]: `(⊤, ⊥, ∧, ∨, →)`

Now we can check if a model or interpretation, represented by a `TruthDict`, satisfies a `Formula`!

In [20]:
```
check(p, soul)  # soul is alive
```
Out[20]:
```
true
```
In [21]:
```
check(φ, soul)  # but not both alive and mortal
```
Out[21]:
```
false
```
In [22]:
```
check(φ, body)  # body is both alive and mortal!
```
Out[22]:
```
true
```
These objects can actually be used as dictionaries from `Formula` to `Truth` values.

For example, we both assign `⊤` (top) to the atom "alive".

In [23]:
```
soul[p], body[p]
```
Out[23]:
```
(⊤, ⊤)
```
Not only, indexing can be used to check generic `Formula` s.

In [24]:
```
soul[φ]
```
Out[24]:
```
⊥
```
This is syntactic sugar for the *interpretation* algorithm, which is actually more general than check!

In [25]:
```
interpret(φ, soul)
```
Out[25]:
```
⊥
```
In fact, it also works under incomplete information.

In [26]:
```
body[φ ∧ Atom("?Unknown property?")]
```
Out[26]:
```
Atom{String}: ?Unknown property?
```
Notice how in this example, with an *unknown atom*, it uses the *known* information to simplify the formula

In [27]:
```
body[φ ∨ Atom("?Unknown property?")]
```
Out[27]:
```
⊤
```
So ultimately, `check` is just a shortcut for making sure that `interpret` simplifies the formula to `⊤`.

In [28]:
```
check(φ, soul) == istop(interpret(φ, soul))
```
Out[28]:
```
true
```

Let's generate random formulas!

In [29]:
```
SoleLogics.BASE_PROPOSITIONAL_CONNECTIVES
```

Out[29]:
```
4-element Vector{NamedConnective}:
 ¬
 ∧
 ∨
 →
```

In [30]:
```
Σ = @atoms a b
```

Out[30]:
```
2-element Vector{Atom{String}}:
 Atom{String}: a
 Atom{String}: b
```

In [31]:
```
using Random

h = 2;    # the height of the formula

φ = randformula(
    Random.MersenneTwister(300),     # the random number generator we want
    h,
    Σ,
    SoleLogics.BASE_PROPOSITIONAL_CONNECTIVES
)
```

Out[31]:
```
SyntaxBranch: ¬b → b → a
```

In [32]:
```
syntaxstring(φ; parenthesize_commutatives = true)
```

Out[32]:
```
"¬b → b → a"
```

In [33]:
```
normalize(φ)
```

Out[33]:
```
SyntaxBranch: ¬b → b → a
```

Exercise: Check many, randomly-generated formulas on the alphabet p, q on both `soul` and `body`. Do `soul` and `body` have the same probability of satisfying a generic formula? Can you estimate this probability?

Scalar Interpretations

Now, let's consider a propositional interpretation on scalar attributes $A_1, A_2, \ldots$, and check formulas on an alphabet $\mathcal{A} \subseteq \{A_i < v, v \in \mathbb{R}\}$ on it.

We start by defining the atoms of type $A_i < v$.

In [34]:
```
import SoleLogics: syntaxstring

struct ConditionOnAttribute
    i_attribute::Integer
    threshold::Real
end

function syntaxstring(c::ConditionOnAttribute; kwargs...)
    "A$(c.i_attribute) < $(c.threshold)"
```

```
end

syntaxstring(ConditionOnAttribute(2, 10))
```

Out[34]: "A2 < 10"

In [35]:
```
using SoleLogics: AbstractAssignment # Abstract type for Interpretations

struct TabularInterpretation{T<:Real} <: AbstractAssignment
    vals::Vector{T}
end

import SoleLogics: interpret, value

function interpret(a::Atom{ConditionOnAttribute}, I::TabularInterpretatio
    cond = value(a)
    return (I.vals[cond.i_attribute] < cond.threshold ? ⊤ : ⊥)
end
```

Out[35]: interpret (generic function with 16 methods)

In [36]:
```
rng = Random.MersenneTwister(1)
n_variables = 4

vals = rand(rng, n_variables)
I = TabularInterpretation(vals)
```

Out[36]: TabularInterpretation{Float64}([0.09913970137863681, 0.7019797138879542, 0.503261785841856, 0.8758412053070399])

In [37]:
```
A = Atom.(
    [ConditionOnAttribute(v, t) for v in 1:n_variables for t in 0:0.1:1.0
)
syntaxstring.(A)
```

Out[37]: 44-element Vector{String}:

```
 "A1 < 0.0"
 "A1 < 0.1"
 "A1 < 0.2"
 "A1 < 0.3"
 "A1 < 0.4"
 "A1 < 0.5"
 "A1 < 0.6"
 "A1 < 0.7"
 "A1 < 0.8"
 "A1 < 0.9"
 "A1 < 1.0"
 "A2 < 0.0"
 "A2 < 0.1"
 ⋮
 "A3 < 1.0"
 "A4 < 0.0"
 "A4 < 0.1"
 "A4 < 0.2"
 "A4 < 0.3"
 "A4 < 0.4"
 "A4 < 0.5"
 "A4 < 0.6"
 "A4 < 0.7"
 "A4 < 0.8"
 "A4 < 0.9"
 "A4 < 1.0"
```

In [38]:

```
[interpret(cond, I) for cond in A]
```

Out[38]: 44-element Vector{BooleanTruth}:

```
 ⊥
 ⊤
 ⊤
 ⊤
 ⊤
 ⊤
 ⊤
 ⊤
 ⊤
 ⊤
 ⊤
 ⊥
 ⊥
 ⋮
 ⊤
 ⊥
 ⊥
 ⊥
 ⊥
 ⊥
 ⊥
 ⊥
 ⊥
 ⊥
 ⊤
 ⊤
```

In [39]:

```
rng = Random.MersenneTwister(32)
```

```
[
    begin
        f = randformula(rng, 3, A, SoleLogics.BASE_PROPOSITIONAL_CONNECTI
        syntaxstring(f) => interpret(f, I)
    end for _ in 1:10
]
```

Out[39]:
```
10-element Vector{Pair{String, BooleanTruth}}:
 "A3 < 0.0" => ⊥
                                                   "A3 < 0.5 ∧ A2 < 0.0 ∧
A4 < 0.4 ∧ A1 < 0.7" => ⊥
 "A4 < 0.4" => ⊥
 "A3 < 1.0" => ⊤
                     "(A4 < 0.0 → A3 < 0.3) ∧ ¬A2 < 0.7 → (A4 < 0.9 → A3
< 0.8) ∧ ¬A1 < 0.7" => ⊥
 "¬A2 < 0.9" => ⊥
  "(A1 < 0.7 → A4 < 0.9) ∨ A2 < 0.6 ∨ A4 < 0.3 ∨ (A1 < 0.5 ∧ A1 < 0.3 → A
1 < 0.9 ∨ A3 < 0.1)" => ⊤
 "A1 < 0.9" => ⊤
 "A3 < 0.9" => ⊤
 "¬¬A1 < 1.0" => ⊤
```

Exercise: Check many, randomly-generated formulas on many, randomly-generated tabular interpretations, and store the formulas that satisfy the highest number of instances!

Modal Logic K

Let's instantiate a frame with 5 worlds and 5 edges.

In [40]:
```
using Graphs

worlds = SoleLogics.World.(1:5)
edges = Edge.([(1,2), (1,3), (2,4), (3,4), (3,5)])
graph = Graphs.SimpleDiGraph(edges)
fr = SoleLogics.ExplicitCrispUniModalFrame(worlds, graph)
```

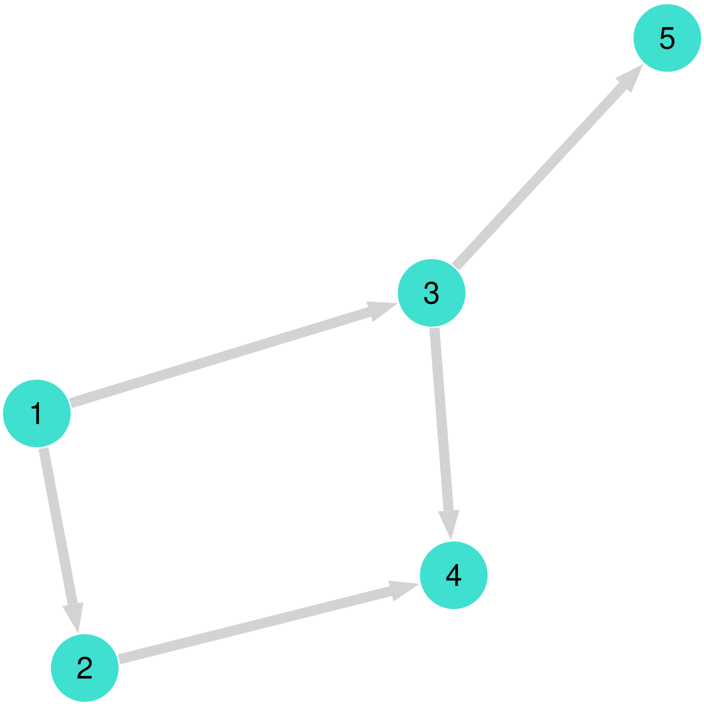
Out[40]:
```
SimpleModalFrame{World{Int64}, SimpleDiGraph{Int64}} with 5 worlds and 5
edges:
- worlds: [1, 2, 3, 4, 5]
- accessibles:
        1 -> [2, 3]
        2 -> [4]
        3 -> [4, 5]
        4 -> []
        5 -> []
```

Let's print the accessibility relation between our worlds.

In [41]:
```
using GraphPlot
import Cairo, Fontconfig
```

```
nodelabel = [i for (i, _) in enumerate(worlds)]
gplot(graph, nodelabel=nodelabel)
```

Out[41]:



We pick the first world...

In [42]:
```
w1 = worlds[1]
```

Out[42]:
```
World{Int64}(1)
```

...and we enumerate the worlds that are accessible from the first world.

In [43]:
```
accessibles(fr, w1)
```

Out[43]:
```
2-element Vector{World{Int64}}:
 World{Int64}(2)
 World{Int64}(3)
```

Let's assign each world a propositional interpretation.

In [44]:
```
valuation = Dict([
    worlds[1] => TruthDict([p => true, q => false]),
    worlds[2] => TruthDict([p => true, q => true]),
    worlds[3] => TruthDict([p => true, q => false]),
    worlds[4] => TruthDict([p => false, q => false]),
    worlds[5] => TruthDict([p => false, q => true]),
]);
```

We instantiate a Kripke structure by combining a Kripke frame and the propositional interpretations over each world.

In [45]:
```
K = KripkeStructure(fr, valuation)
```

```
Out[45]: KripkeStructure{SimpleModalFrame{World{Int64}, SimpleDiGraph{Int64}}, Di
ct{World{Int64}, TruthDict{Dict{Atom{String}, BooleanTruth}}}} with
- frame: SimpleModalFrame{World{Int64}, SimpleDiGraph{Int64}} with 5 wor
lds and 5 edges:
    - worlds: [1, 2, 3, 4, 5]
    - accessibles:
        1 -> [2, 3]
        2 -> [4]
        3 -> [4, 5]
        4 -> []
        5 -> []
- valuations:
        1 -> TruthDict([it's mortal! => ⊥, it's alive! => ⊤])
        2 -> TruthDict([it's mortal! => ⊤, it's alive! => ⊤])
        3 -> TruthDict([it's mortal! => ⊥, it's alive! => ⊤])
        4 -> TruthDict([it's mortal! => ⊥, it's alive! => ⊥])
        5 -> TruthDict([it's mortal! => ⊤, it's alive! => ⊥])
```

Let's generate a random modal formula!

```
In [46]: SoleLogics.BASE_MODAL_CONNECTIVES
```

```
Out[46]: 6-element Vector{NamedConnective}:
 ¬
 ∧
 ∨
 →
 ◊
 □
```

```
In [47]: φmodal = randformula(
             Random.MersenneTwister(5678),
             3,      # height
             [p,q],  # alphabet
             SoleLogics.BASE_MODAL_CONNECTIVES
         )
```

```
Out[47]: SyntaxBranch: ¬it's mortal!
```

```
In [48]: normalize(φmodal)
```

```
Out[48]: SyntaxBranch: ¬it's mortal!
```

Let's check the formula on each world of the Kripke structure!

```
In [49]: [w => check(φmodal, K, w) for w in worlds]
```

```
Out[49]: 5-element Vector{Pair{World{Int64}, Bool}}:
 World{Int64}(1) => 1
 World{Int64}(2) => 0
 World{Int64}(3) => 1
 World{Int64}(4) => 1
 World{Int64}(5) => 0
```

Exercise: check many, randomly-generated *modal* formulas on many, randomly-generated *modal* interpretations, and store the formulas that satisfy the highest number of instances!

Exercise: define a structure for representing a *modal* interpretation on scalar variables. You can see if your solution it works by running the cell below, which instantiates a random modal scalar interpretation. *Remember to remove* `#=` *and* `=#` *to uncomment the following two cells!*

In [50]:
```
#=
module exercise4

export ModalInterpretation

using Main: ConditionOnAttribute

using SoleLogics

using SoleLogics: AbstractFrame, World, AbstractKripkeStructure

# TODO:
# struct ModalInterpretation{F
#     R<:AbstractFrame,
#     T<:Real
# } <: AbstractKripkeStructure
#     frame::FR
#     vals::???
# end

import SoleLogics: interpret, frame

# Retrieve the interpretation's frame
frame(i::ModalInterpretation) = i.frame

# TODO:
# function interpret(
#     a::Atom{ConditionOnAttribute},
#     I::ModalInterpretation,
#     w::World
# )
#     cond = value(a)
#     v = ???
#     return (v < cond.threshold ? TOP : BOT)
# end


end # end module
=#
```

In [51]:
```
#=
using .exercise4

rng = Random.MersenneTwister(1)
n_variables = 4
n_worlds = 5
n_edges = 7
n_formulas = 10

worlds = SoleLogics.World.(1:n_worlds)
g = SimpleDiGraph(n_worlds, n_edges; rng)
fr = SoleLogics.ExplicitCrispUniModalFrame(worlds, g)
variable_values = [rand(n_variables) for w in worlds]
```

```
Imodal = ModalInterpretation(fr, variable_values)

for i_formula in 1:n_formulas
    φmodal = randformula(
        Random.MersenneTwister(i_formula),
        2,
        A,
        SoleLogics.BASE_MODAL_CONNECTIVES
    )

    println(
        syntaxstring(φmodal) => [
            "w$(SoleLogics.name(w))" => check(φmodal, Imodal, w) for w in
        ]
    )
    println()
end
=#
```

Multi-Modal Logic (i.e., Temporal and Spatial Logics)

`SoleLogics.jl` provides useful aliases for the most common multi-modal logics:

- Linear Temporal Logic with Future and Past (LTL[F,P])
- Compass Logic (CL)
- Halpern and Shoham's Modal Logic of Time Intervals (HS)
- Lutz and Wolter's Modal Logic of Topological Relations with rectangular areas aligned with the axes (LRCC8_Rec)

Relation names are given in an uniform way as follows:
`NAMEOFTHELOGIC` _ `NAMEOFTHERELATION`

For example, the relation Future (F) for LTL[F,P] will be named: `LTLFP_F`

Linear Temporal Logic with Future and Past (LTL[F,P])

In [52]:
```
using SoleLogics: LTLFP_F, LTLFP_P  # These aliases are not exported by d
```

Let's begin by defining a set of 5 ordered points in time.

In [53]:
```
fr = FullDimensionalFrame((5,), Point1D{Int})
allworlds(fr) |> collect
```

Out[53]:
```
5-element Vector{Point1D{Int64}}:
 Point1D{Int64}((1,))
 Point1D{Int64}((2,))
 Point1D{Int64}((3,))
 Point1D{Int64}((4,))
 Point1D{Int64}((5,))
```

Which points are in the future, w.r.t. `3` ?

In [54]:
```
collect(accessibles(fr, Point(3), LTLFP_F))
```

Out[54]:
```
2-element Vector{Point1D{Int64}}:
 Point1D{Int64}((4,))
 Point1D{Int64}((5,))
```

Which points are in the past, w.r.t. `3` ?

In [55]:
```
collect(accessibles(fr, Point(3), LTLFP_P))
```

Out[55]:
```
2-element Vector{Point1D{Int64}}:
 Point1D{Int64}((1,))
 Point1D{Int64}((2,))
```

Is `5` happening later than `2` ?

In [56]:
```
Point(5) in accessibles(fr, Point(2), LTLFP_F)
```

Out[56]:
```
true
```

Is `1` happening later than `2` ?

In [57]:
```
Point(1) in accessibles(fr, Point(2), LTLFP_F)
```

Out[57]:
```
false
```

How many points are later than `2` (over the 5 points we have defined)?

In [58]:
```
length(accessibles(fr, Point(2), LTLFP_F))
```

Out[58]:
```
3
```

Compass Logic (CL)

In [59]:
```
using SoleLogics: CL_N, CL_S, CL_E, CL_W
```

Let's define a set of 25 points, of coordinates `(x,y)`, in space.

In [60]:
```
fr = FullDimensionalFrame((5,5), Point2D{Int})
allworlds(fr) |> collect
```

Out[60]:
```
5×5 Matrix{Point2D{Int64}}:
 Point2D{Int64}((1, 1))  Point2D{Int64}((1, 2))  …  Point2D{Int64}((1, 5))
 Point2D{Int64}((2, 1))  Point2D{Int64}((2, 2))     Point2D{Int64}((2, 5))
 Point2D{Int64}((3, 1))  Point2D{Int64}((3, 2))     Point2D{Int64}((3, 5))
 Point2D{Int64}((4, 1))  Point2D{Int64}((4, 2))     Point2D{Int64}((4, 5))
 Point2D{Int64}((5, 1))  Point2D{Int64}((5, 2))     Point2D{Int64}((5, 5))
```

Which points are north of `(3,3)` ?

In [61]:
```
collect(accessibles(fr, Point(3,3), CL_N))
```

Out[61]:
```
2-element Vector{Point2D{Int64}}:
 Point2D{Int64}((3, 4))
 Point2D{Int64}((3, 5))
```

Which points are south of `(3,3)` ?

In [62]:
```
collect(accessibles(fr, Point(3,3), CL_S))
```

Out[62]:
```
2-element Vector{Point2D{Int64}}:
 Point2D{Int64}((3, 1))
 Point2D{Int64}((3, 2))
```

Which points are east of `(3,3)` ?

In [63]:
```
collect(accessibles(fr, Point(3,3), CL_E))
```

Out[63]:
```
2-element Vector{Point2D{Int64}}:
 Point2D{Int64}((4, 3))
 Point2D{Int64}((5, 3))
```

Which points are west of `(3,3)` ?

In [64]:
```
collect(accessibles(fr, Point(3,3), CL_W))
```

Out[64]:
```
2-element Vector{Point2D{Int64}}:
 Point2D{Int64}((1, 3))
 Point2D{Int64}((2, 3))
```

Is `(2,4)` west of `(5,4)` ?

In [65]:
```
Point(2,4) in accessibles(fr, Point(5,4), CL_W)
```

Out[65]:
```
true
```

Is `(2,4)` north of `(1,3)` ?

In [66]:
```
Point(2,4) in accessibles(fr, Point(1,3), CL_N)
```

Out[66]:
```
false
```

Is `(2,4)` north of `(2,3)` ?

In [67]:
```
Point(2,4) in accessibles(fr, Point(2,3), CL_N)
```

Out[67]:
```
true
```

How many points are north of `(4,2)` ?

In [68]:
```
length(accessibles(fr, Point(4,2), CL_N))
```

Out[68]:
```
3
```

Halpern and Shoham's Modal Logic of Time Intervals (HS)

Remember: all relations are mutually disjoint!

(i.e., only one is valid at one time).

In [69]:
```
using SoleLogics: HS_A, HS_L, HS_B, HS_E, HS_D, HS_O
using SoleLogics: HS_Ai, HS_Li, HS_Bi, HS_Ei, HS_Di, HS_Oi
```

Let's generate 10 linearly spaced intervals; to do so, we need 11 points.

Combining them we can get up to 55 intervals!

In [70]:
```
fr = FullDimensionalFrame((10,), SoleLogics.Interval{Int})
allworlds(fr) |> collect
```

Out[70]:
```
55-element Vector{Interval{Int64}}:
 Interval{Int64}(1, 2)
 Interval{Int64}(1, 3)
 Interval{Int64}(2, 3)
 Interval{Int64}(1, 4)
 Interval{Int64}(2, 4)
 Interval{Int64}(3, 4)
 Interval{Int64}(1, 5)
 Interval{Int64}(2, 5)
 Interval{Int64}(3, 5)
 Interval{Int64}(4, 5)
 Interval{Int64}(1, 6)
 Interval{Int64}(2, 6)
 Interval{Int64}(3, 6)
 ⋮
 Interval{Int64}(8, 10)
 Interval{Int64}(9, 10)
 Interval{Int64}(1, 11)
 Interval{Int64}(2, 11)
 Interval{Int64}(3, 11)
 Interval{Int64}(4, 11)
 Interval{Int64}(5, 11)
 Interval{Int64}(6, 11)
 Interval{Int64}(7, 11)
 Interval{Int64}(8, 11)
 Interval{Int64}(9, 11)
 Interval{Int64}(10, 11)
```

Which intervals come after `[4,8]` ?

In [71]:
```
collect(accessibles(fr, SoleLogics.Interval(4,8), HS_A))
```

Out[71]:
```
3-element Vector{Interval{Int64}}:
 Interval{Int64}(8, 9)
 Interval{Int64}(8, 10)
 Interval{Int64}(8, 11)
```

Which come later than `[4,8]` ?

In [72]:
```
collect(accessibles(fr, SoleLogics.Interval(4,8), HS_L))
```

Out[72]:
```
3-element Vector{Interval{Int64}}:
 Interval{Int64}(9, 10)
 Interval{Int64}(9, 11)
 Interval{Int64}(10, 11)
```

Which intervals begin with `[4,8]`, but end before?

In [73]:
```
collect(accessibles(fr, SoleLogics.Interval(4,8), HS_B))
```

Out[73]:
```
3-element Vector{Interval{Int64}}:
 Interval{Int64}(4, 5)
 Interval{Int64}(4, 6)
 Interval{Int64}(4, 7)
```

Which intervals begin after `[4,8]`, but end with the same point?

In [74]:
```
collect(accessibles(fr, SoleLogics.Interval(4,8), HS_E))
```

Out[74]:
```
3-element Vector{Interval{Int64}}:
 Interval{Int64}(5, 8)
 Interval{Int64}(6, 8)
 Interval{Int64}(7, 8)
```

Which intervals happen during `[4,8]`?

In [75]:
```
collect(accessibles(fr, SoleLogics.Interval(4,8), HS_D))
```

Out[75]:
```
3-element Vector{Interval{Int64}}:
 Interval{Int64}(5, 6)
 Interval{Int64}(5, 7)
 Interval{Int64}(6, 7)
```

Which intervals overlap with `[4,8]` but begin later?

In [76]:
```
collect(accessibles(fr, SoleLogics.Interval(4,8), HS_O))
```

Out[76]:
```
9-element Vector{Interval{Int64}}:
 Interval{Int64}(5, 9)
 Interval{Int64}(5, 10)
 Interval{Int64}(5, 11)
 Interval{Int64}(6, 9)
 Interval{Int64}(6, 10)
 Interval{Int64}(6, 11)
 Interval{Int64}(7, 9)
 Interval{Int64}(7, 10)
 Interval{Int64}(7, 11)
```

Which intervals end exactly when `[4,8]` begins?

In [77]:
```
collect(accessibles(fr, SoleLogics.Interval(4,8), HS_Ai))
```

Out[77]:
```
3-element Vector{Interval{Int64}}:
 Interval{Int64}(1, 4)
 Interval{Int64}(2, 4)
 Interval{Int64}(3, 4)
```

Which intervals happen before `[4,8]`?

In [78]:
```
collect(accessibles(fr, SoleLogics.Interval(4,8), HS_Li))
```

```
Out[78]: 3-element Vector{Interval{Int64}}:
 Interval{Int64}(1, 2)
 Interval{Int64}(1, 3)
 Interval{Int64}(2, 3)
```

Which intervals begin with `[4,8]`, but end later?

```julia
In [79]: collect(accessibles(fr, SoleLogics.Interval(4,8), HS_Bi))
```

```
Out[79]: 3-element Vector{Interval{Int64}}:
 Interval{Int64}(4, 9)
 Interval{Int64}(4, 10)
 Interval{Int64}(4, 11)
```

Which intervals end with `[4,8]`, but start before?

```julia
In [80]: collect(accessibles(fr, SoleLogics.Interval(4,8), HS_Ei))
```

```
Out[80]: 3-element Vector{Interval{Int64}}:
 Interval{Int64}(1, 8)
 Interval{Int64}(2, 8)
 Interval{Int64}(3, 8)
```

Which intervals contain `[4,8]`?

```julia
In [81]: collect(accessibles(fr, SoleLogics.Interval(4,8), HS_Di))
```

```
Out[81]: 9-element Vector{Interval{Int64}}:
 Interval{Int64}(1, 9)
 Interval{Int64}(1, 10)
 Interval{Int64}(1, 11)
 Interval{Int64}(2, 9)
 Interval{Int64}(2, 10)
 Interval{Int64}(2, 11)
 Interval{Int64}(3, 9)
 Interval{Int64}(3, 10)
 Interval{Int64}(3, 11)
```

Which intervals overlap with `[4,8]` but begin before?

```julia
In [82]: collect(accessibles(fr, SoleLogics.Interval(4,8), HS_Oi))
```

```
Out[82]: 9-element Vector{Interval{Int64}}:
 Interval{Int64}(1, 5)
 Interval{Int64}(1, 6)
 Interval{Int64}(1, 7)
 Interval{Int64}(2, 5)
 Interval{Int64}(2, 6)
 Interval{Int64}(2, 7)
 Interval{Int64}(3, 5)
 Interval{Int64}(3, 6)
 Interval{Int64}(3, 7)
```

Lutz and Wolter's Modal Logic of Topological Relations with rectangular areas aligned with the axes (LRCC8_Rec)

```julia
In [83]: using SoleLogics: LRCC8_Rec_DC, LRCC8_Rec_EC, LRCC8_Rec_PO
         using SoleLogics: LRCC8_Rec_TPP, LRCC8_Rec_TPPi, LRCC8_Rec_NTPP, LRCC8_Re
```

Let's create 5 linearly spaced intervals on both axes (i.e., 6 points on each).

We can combine them in 225 different ways to define rectangles!

Note that rectangles are specified using a tuple of intervals, one on each axis, e.g., `([2,4], [3,5])` identifies a rectangle using `Point`s `2` and `4` from the `x` axis and `3` and `5` from the `y` axis.

In [84]:
```
fr = FullDimensionalFrame((5,5), SoleLogics.Interval2D{Int})
allworlds(fr) |> collect
```

Out[84]:
```
225-element Vector{Interval2D{Int64}}:
 Interval2D{Int64}(Interval{Int64}(1, 2), Interval{Int64}(1, 2))
 Interval2D{Int64}(Interval{Int64}(1, 3), Interval{Int64}(1, 2))
 Interval2D{Int64}(Interval{Int64}(2, 3), Interval{Int64}(1, 2))
 Interval2D{Int64}(Interval{Int64}(1, 4), Interval{Int64}(1, 2))
 Interval2D{Int64}(Interval{Int64}(2, 4), Interval{Int64}(1, 2))
 Interval2D{Int64}(Interval{Int64}(3, 4), Interval{Int64}(1, 2))
 Interval2D{Int64}(Interval{Int64}(1, 5), Interval{Int64}(1, 2))
 Interval2D{Int64}(Interval{Int64}(2, 5), Interval{Int64}(1, 2))
 Interval2D{Int64}(Interval{Int64}(3, 5), Interval{Int64}(1, 2))
 Interval2D{Int64}(Interval{Int64}(4, 5), Interval{Int64}(1, 2))
 Interval2D{Int64}(Interval{Int64}(1, 6), Interval{Int64}(1, 2))
 Interval2D{Int64}(Interval{Int64}(2, 6), Interval{Int64}(1, 2))
 Interval2D{Int64}(Interval{Int64}(3, 6), Interval{Int64}(1, 2))
 ⋮
 Interval2D{Int64}(Interval{Int64}(1, 4), Interval{Int64}(5, 6))
 Interval2D{Int64}(Interval{Int64}(2, 4), Interval{Int64}(5, 6))
 Interval2D{Int64}(Interval{Int64}(3, 4), Interval{Int64}(5, 6))
 Interval2D{Int64}(Interval{Int64}(1, 5), Interval{Int64}(5, 6))
 Interval2D{Int64}(Interval{Int64}(2, 5), Interval{Int64}(5, 6))
 Interval2D{Int64}(Interval{Int64}(3, 5), Interval{Int64}(5, 6))
 Interval2D{Int64}(Interval{Int64}(4, 5), Interval{Int64}(5, 6))
 Interval2D{Int64}(Interval{Int64}(1, 6), Interval{Int64}(5, 6))
 Interval2D{Int64}(Interval{Int64}(2, 6), Interval{Int64}(5, 6))
 Interval2D{Int64}(Interval{Int64}(3, 6), Interval{Int64}(5, 6))
 Interval2D{Int64}(Interval{Int64}(4, 6), Interval{Int64}(5, 6))
 Interval2D{Int64}(Interval{Int64}(5, 6), Interval{Int64}(5, 6))
```

How many rectangles can I build so that they are disconnected from `([3,4], [3,4])`?

In [85]:
```
length(
    collect(accessibles(fr, SoleLogics.Interval2D((3,4),(3,4)), LRCC8_Rec
)
```

Out[85]: 56

How many rectangles can I define so that they are inside `([2,5], [2,5])` and touching it on the perimeter? (i.e., tangential proper part)

In [86]:
```
length(
    collect(accessibles(fr, SoleLogics.Interval2D((2,5),(2,5)), LRCC8_Rec
)
```

Out[86]: 34

How many rectangles can I define so that they contain `([3,4], [3,4])` not touching it on the perimeter? (i.e., non-tangential proper part inverse)

In [87]:
```
length(
    collect(
        accessibles(fr, SoleLogics.Interval2D((3,4),(3,4)), LRCC8_Rec_NTP
    )
)
```

Out[87]:
```
16
```

Let's create another frame consisting of 3 linearly spaced intervals on both axes (i.e., 4 points on each).

We can combine them in 36 different ways to define rectangles!

In [88]:
```
fr = FullDimensionalFrame((3,3), SoleLogics.Interval2D{Int})
allworlds(fr) |> collect
```

Out[88]:
```
36-element Vector{Interval2D{Int64}}:
 Interval2D{Int64}(Interval{Int64}(1, 2), Interval{Int64}(1, 2))
 Interval2D{Int64}(Interval{Int64}(1, 3), Interval{Int64}(1, 2))
 Interval2D{Int64}(Interval{Int64}(2, 3), Interval{Int64}(1, 2))
 Interval2D{Int64}(Interval{Int64}(1, 4), Interval{Int64}(1, 2))
 Interval2D{Int64}(Interval{Int64}(2, 4), Interval{Int64}(1, 2))
 Interval2D{Int64}(Interval{Int64}(3, 4), Interval{Int64}(1, 2))
 Interval2D{Int64}(Interval{Int64}(1, 2), Interval{Int64}(1, 3))
 Interval2D{Int64}(Interval{Int64}(1, 3), Interval{Int64}(1, 3))
 Interval2D{Int64}(Interval{Int64}(2, 3), Interval{Int64}(1, 3))
 Interval2D{Int64}(Interval{Int64}(1, 4), Interval{Int64}(1, 3))
 Interval2D{Int64}(Interval{Int64}(2, 4), Interval{Int64}(1, 3))
 Interval2D{Int64}(Interval{Int64}(3, 4), Interval{Int64}(1, 3))
 Interval2D{Int64}(Interval{Int64}(1, 2), Interval{Int64}(2, 3))
 ⋮
 Interval2D{Int64}(Interval{Int64}(1, 2), Interval{Int64}(2, 4))
 Interval2D{Int64}(Interval{Int64}(1, 3), Interval{Int64}(2, 4))
 Interval2D{Int64}(Interval{Int64}(2, 3), Interval{Int64}(2, 4))
 Interval2D{Int64}(Interval{Int64}(1, 4), Interval{Int64}(2, 4))
 Interval2D{Int64}(Interval{Int64}(2, 4), Interval{Int64}(2, 4))
 Interval2D{Int64}(Interval{Int64}(3, 4), Interval{Int64}(2, 4))
 Interval2D{Int64}(Interval{Int64}(1, 2), Interval{Int64}(3, 4))
 Interval2D{Int64}(Interval{Int64}(1, 3), Interval{Int64}(3, 4))
 Interval2D{Int64}(Interval{Int64}(2, 3), Interval{Int64}(3, 4))
 Interval2D{Int64}(Interval{Int64}(1, 4), Interval{Int64}(3, 4))
 Interval2D{Int64}(Interval{Int64}(2, 4), Interval{Int64}(3, 4))
 Interval2D{Int64}(Interval{Int64}(3, 4), Interval{Int64}(3, 4))
```

How many rectangles are externally connected with `([2,3],[2,3])`?

In [89]:
```
length(
    collect(accessibles(fr, SoleLogics.Interval2D((2,3),(2,3)), LRCC8_Rec_
)
```

Out[89]:
```
20
```

Let's create another frame consisting of 4 linearly spaced intervals on both axes (i.e., 5 points on each).

We can combine them in 100 different ways to define rectangles!

In [90]:
```
fr = FullDimensionalFrame((4,4), SoleLogics.Interval2D{Int})
allworlds(fr) |> collect
```

Out[90]:
```
100-element Vector{Interval2D{Int64}}:
 Interval2D{Int64}(Interval{Int64}(1, 2), Interval{Int64}(1, 2))
 Interval2D{Int64}(Interval{Int64}(1, 3), Interval{Int64}(1, 2))
 Interval2D{Int64}(Interval{Int64}(2, 3), Interval{Int64}(1, 2))
 Interval2D{Int64}(Interval{Int64}(1, 4), Interval{Int64}(1, 2))
 Interval2D{Int64}(Interval{Int64}(2, 4), Interval{Int64}(1, 2))
 Interval2D{Int64}(Interval{Int64}(3, 4), Interval{Int64}(1, 2))
 Interval2D{Int64}(Interval{Int64}(1, 5), Interval{Int64}(1, 2))
 Interval2D{Int64}(Interval{Int64}(2, 5), Interval{Int64}(1, 2))
 Interval2D{Int64}(Interval{Int64}(3, 5), Interval{Int64}(1, 2))
 Interval2D{Int64}(Interval{Int64}(4, 5), Interval{Int64}(1, 2))
 Interval2D{Int64}(Interval{Int64}(1, 2), Interval{Int64}(1, 3))
 Interval2D{Int64}(Interval{Int64}(1, 3), Interval{Int64}(1, 3))
 Interval2D{Int64}(Interval{Int64}(2, 3), Interval{Int64}(1, 3))
 ⋮
 Interval2D{Int64}(Interval{Int64}(3, 5), Interval{Int64}(3, 5))
 Interval2D{Int64}(Interval{Int64}(4, 5), Interval{Int64}(3, 5))
 Interval2D{Int64}(Interval{Int64}(1, 2), Interval{Int64}(4, 5))
 Interval2D{Int64}(Interval{Int64}(1, 3), Interval{Int64}(4, 5))
 Interval2D{Int64}(Interval{Int64}(2, 3), Interval{Int64}(4, 5))
 Interval2D{Int64}(Interval{Int64}(1, 4), Interval{Int64}(4, 5))
 Interval2D{Int64}(Interval{Int64}(2, 4), Interval{Int64}(4, 5))
 Interval2D{Int64}(Interval{Int64}(3, 4), Interval{Int64}(4, 5))
 Interval2D{Int64}(Interval{Int64}(1, 5), Interval{Int64}(4, 5))
 Interval2D{Int64}(Interval{Int64}(2, 5), Interval{Int64}(4, 5))
 Interval2D{Int64}(Interval{Int64}(3, 5), Interval{Int64}(4, 5))
 Interval2D{Int64}(Interval{Int64}(4, 5), Interval{Int64}(4, 5))
```

How many rectangles are partially overlapping with `([2,4],[2,4])` ?

In [91]:
```
length(
    collect(accessibles(fr, SoleLogics.Interval2D((2,4),(2,4)), LRCC8_Rec
)
```

Out[91]:
```
40
```

How many rectangles contain `([2,4],[2,4])` and touch it on the perimeter?

In [92]:
```
length(
    collect(accessibles(fr, SoleLogics.Interval2D((2,4),(2,4)), LRCC8_Rec
)
```

Out[92]:
```
14
```

Let's create a final frame consisting of 6 linearly spaced intervals on both axes (i.e., 7 points on each).

We can combine them in 441 different ways to define rectangles!

In [93]:
```
fr = FullDimensionalFrame((6,6), SoleLogics.Interval2D{Int})
allworlds(fr) |> collect
```

```
Out[93]: 441-element Vector{Interval2D{Int64}}:
 Interval2D{Int64}(Interval{Int64}(1, 2), Interval{Int64}(1, 2))
 Interval2D{Int64}(Interval{Int64}(1, 3), Interval{Int64}(1, 2))
 Interval2D{Int64}(Interval{Int64}(2, 3), Interval{Int64}(1, 2))
 Interval2D{Int64}(Interval{Int64}(1, 4), Interval{Int64}(1, 2))
 Interval2D{Int64}(Interval{Int64}(2, 4), Interval{Int64}(1, 2))
 Interval2D{Int64}(Interval{Int64}(3, 4), Interval{Int64}(1, 2))
 Interval2D{Int64}(Interval{Int64}(1, 5), Interval{Int64}(1, 2))
 Interval2D{Int64}(Interval{Int64}(2, 5), Interval{Int64}(1, 2))
 Interval2D{Int64}(Interval{Int64}(3, 5), Interval{Int64}(1, 2))
 Interval2D{Int64}(Interval{Int64}(4, 5), Interval{Int64}(1, 2))
 Interval2D{Int64}(Interval{Int64}(1, 6), Interval{Int64}(1, 2))
 Interval2D{Int64}(Interval{Int64}(2, 6), Interval{Int64}(1, 2))
 Interval2D{Int64}(Interval{Int64}(3, 6), Interval{Int64}(1, 2))
 ⋮
 Interval2D{Int64}(Interval{Int64}(4, 5), Interval{Int64}(6, 7))
 Interval2D{Int64}(Interval{Int64}(1, 6), Interval{Int64}(6, 7))
 Interval2D{Int64}(Interval{Int64}(2, 6), Interval{Int64}(6, 7))
 Interval2D{Int64}(Interval{Int64}(3, 6), Interval{Int64}(6, 7))
 Interval2D{Int64}(Interval{Int64}(4, 6), Interval{Int64}(6, 7))
 Interval2D{Int64}(Interval{Int64}(5, 6), Interval{Int64}(6, 7))
 Interval2D{Int64}(Interval{Int64}(1, 7), Interval{Int64}(6, 7))
 Interval2D{Int64}(Interval{Int64}(2, 7), Interval{Int64}(6, 7))
 Interval2D{Int64}(Interval{Int64}(3, 7), Interval{Int64}(6, 7))
 Interval2D{Int64}(Interval{Int64}(4, 7), Interval{Int64}(6, 7))
 Interval2D{Int64}(Interval{Int64}(5, 7), Interval{Int64}(6, 7))
 Interval2D{Int64}(Interval{Int64}(6, 7), Interval{Int64}(6, 7))
```

How many rectangles are contained in `([2,6], [2,6])` without touching it on the perimeter?

```
In [94]: length(
             collect(accessibles(fr, SoleLogics.Interval2D((2,6),(2,6)), LRCC8_Rec
         )
```

```
Out[94]: 9
```

Fuzzy Logic

In standard fuzzy logics, instead of constraining ourselves to only `true` and `false` values, we let them be anythig between the continuous interval [0, 1].

On the downside, we cannot use the classical evaluation for the propositional operators - what does it mean to be `0.3` and `0.5` ?

Fuzzy logics are defined over a `t-norm` operation, which will be our conjunction ($\wedge$); we have 3 of them(*):

- Goedel Logic, where the $tnorm(x, y)$ is defined as the $min\{x, y\}$
- Lukasiewicz Logic, where the $tnorm(x, y)$ is defined as the $max\{0, x + y - 1\}$
- Product Logic, where the $tnorm(x, y)$ is defined as the arithmetic product $x \cdot y$

For each logic, the implication $x \rightarrow y$ will be defined as the $max\{z | tnorm(x, z) \leq y\}$.

This can feel overwhelming at first, as it is very general: the important part is that we can derive the implication for each fuzzy logic from their `t-norm`.

Finally, we will consider the disjunction between two values x and y simply as the $max\{x, y\}$ for all logics.

(*) All other t-norms (hence, fuzzy logics) can be derived though a linear combination of these 3.

In [95]:
```
using SoleLogics.ManyValuedLogics   # fuzzy logics are defined in this su
```

In [96]:
```
GodelLogic
```

Out[96]: FuzzyLogic(t-norm: Gödel (min))

In [97]:
```
bot(GodelLogic)
```

Out[97]: 0.0

In [98]:
```
top(GodelLogic)
```

Out[98]: 1.0

In [99]:
```
unknown = ContinuousTruth(0.5)
```

Out[99]: 0.5

The $tnorm(x, y)$ for Goedel Logic is defined as the $min\{x, y\}$.

In [100…
```
GodelLogic.tnorm(
    unknown,
    unknown
)
```

Out[100… 0.5

In [101…
```
LukasiewiczLogic
```

Out[101… FuzzyLogic(t-norm: Łukasiewicz)

The $tnorm\{x, y\}$ for Lukasiewicz Logic is defined as the $max\{0, x + y - 1\}$.

In [102…
```
LukasiewiczLogic.tnorm(
    unknown,
    unknown
)
```

Out[102… 0.0

The $tnorm\{x, y\}$ for Product Logic is defined as the arithmetic product $x \cdot y$.

In [103…
```
ProductLogic.tnorm(
    unknown,
    unknown
)
```

Out[103… 0.25

Many-Valued Logic

While in fuzzy logics we consider a total order between all the values, many-valued logics go even a step further: we also take into consideration values which can be non-comparable, i.e., partial orders.

To do so, we leverage algebraic structures comprising lattices, such as Heyting Algebras.

In [104…
```
using SoleLogics.ManyValuedLogics: G4, Ł4, H4
using SoleLogics.ManyValuedLogics: α, β
```

In [105…
```
getdomain(G4)
```

Out[105…
```
4-element StaticArraysCore.SVector{4, FiniteTruth} with indices SOneTo
(4):
 ⊤
 ⊥
 α
 β
```

In [106…
```
getdomain(Ł4)
```

Out[106…
```
4-element StaticArraysCore.SVector{4, FiniteTruth} with indices SOneTo
(4):
 ⊤
 ⊥
 α
 β
```

In [107…
```
getdomain(H4)
```

Out[107…
```
4-element StaticArraysCore.SVector{4, FiniteTruth} with indices SOneTo
(4):
 ⊤
 ⊥
 α
 β
```

In [108…
```
precedes(G4, α, β)
```

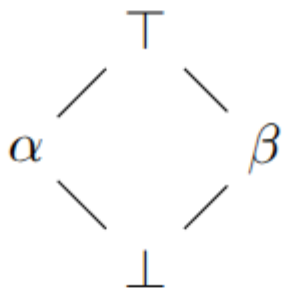
Out[108… true

In [109…
```
precedes(Ł4, α, β)
```

Out[109… true

G4 and Ł4 are fuzzy logics (i.e., they are totally ordered).

The following, on the other hand, is an example of a Heyting Algebra with 4 element, partially ordered.



In [110…
```
precedes(H4, α, β)
```
Out[110…
```
false
```

In [111…
```
precedes(H4, ⊥, α)
```
Out[111…
```
true
```

In [112…
```
precedes(H4, ⊥, β)
```
Out[112…
```
true
```

In [113…
```
precedes(H4, α, ⊤)
```
Out[113…
```
true
```

In [114…
```
precedes(H4, β, ⊤)
```
Out[114…
```
true
```

While fuzzy logics differ on the `t-norm`, many-valued logics in general are defined, among other things (like the set of values), over a more general structure, called a `monoid`, that we will use to interpret conjunction ($\wedge$).

This coincide with the `t-norm` for fuzzy logics.

In [115…
```
G4.monoid(α, β) # min(α, β)
```
Out[115…
```
α
```

In [116…
```
Ł4.monoid(α, β) # max(α+β-1, 0)
```
Out[116…
```
⊥
```

Heyting algebras can be thought of as a generalization of Goedel algebras (resp. Goedel Logic) to partial orders: instead of taking the $min\{x, y\}$, which is not always possible, we take the $inf\{x, y\}$, i.e., the greatest lower bound.

For instance, since in our case α and β are non-comparable, but both greater than $\bot$ (and bigger than $\top$), their inf will be $\bot$.

An inf and a sup is always guaranteed to exist (and to be unique) by definition of a lattice.

In [117…

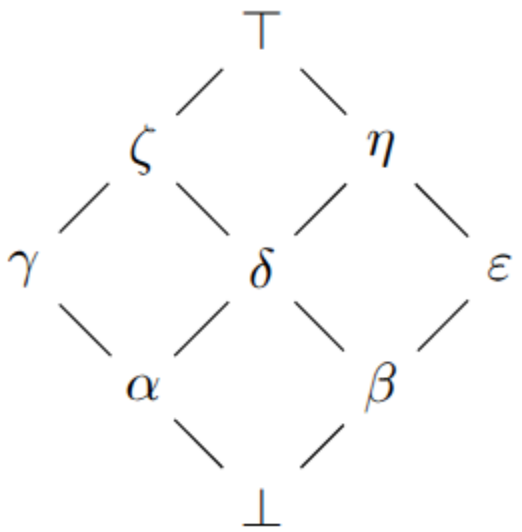
```
H4.monoid(α, β) # inf(α, β)
```

Out[117…

```
⊥
```

Let's have a quick look at a more complex example.

The following is also an Heyting algebras, but with 9 values.



In [118…

```
using SoleLogics.ManyValuedLogics: H9
using SoleLogics.ManyValuedLogics: ζ, η

getdomain(H9)
```

Out[118…

```
9-element StaticArraysCore.SVector{9, FiniteTruth} with indices SOneTo
(9):
 ⊤
 ⊥
 α
 β
 γ
 δ
 ε
 ζ
 η
```

Let's take, for example, values ζ and η: they are non-comparable, but they are both bigger than $\perp, \alpha, \beta, \delta$.

In [119…

```
using SoleLogics.ManyValuedLogics: lesservalues

lesservalues(H9, ζ)
```

Out[119…

```
5-element Vector{FiniteTruth}:
 ⊥
 α
 β
 γ
 δ
```

In [120…
```
lesservalues(H9, η)
```

Out[120…
```
5-element Vector{FiniteTruth}:
 ⊥
 α
 β
 δ
 ε
```

In [121…
```
intersect(lesservalues(H9, ζ), lesservalues(H9, η))
```

Out[121…
```
4-element Vector{FiniteTruth}:
 ⊥
 α
 β
 δ
```

Hence, the greatest of these values (δ) will be our result.

In [122…
```
H9.monoid(ζ, η)
```

Out[122…
```
δ
```

Exercise: try to do the same with other values from the H9 algebra, such as:

- ζ and ϵ
- ζ and β
- α and β

Disjunction ($\vee$) and implication ($\rightarrow$) are generalized in a similar way:

- to evaluate a disjunction $x \vee y$, we will use the $sup\{x, y\}$ (lowest greater bound)
- to evaluate an implication $x \rightarrow y$, we will use the $sup\{z | monoid(x, z) \preceq y\}$

Let's focus on the latter with an example.

In [123…
```
x = ζ
y = η

candidates = Vector{FiniteTruth}()

for z in getdomain(H9)
    r = H9.monoid(x, z)
    print("$x·$z=$r")
    print("\t$r≤$y")
    if precedeq(H9, r, y)
        print("\t✅")
        push!(candidates, z)    # WATCH OUT! I'm pushing z, NOT r!
        println("\tPushing $z to the list of candidates...")
    else
        println("\t❌")
    end
end

println("\ncandidates: $candidates")
```

```
ζ⋅⊤=ζ    ζ≤η      ❌
ζ⋅⊥=⊥    ⊥≤η      ✅       Pushing ⊥ to the list of candidates...
ζ⋅α=α    α≤η      ✅       Pushing α to the list of candidates...
ζ⋅β=β    β≤η      ✅       Pushing β to the list of candidates...
ζ⋅γ=γ    γ≤η      ❌
ζ⋅δ=δ    δ≤η      ✅       Pushing δ to the list of candidates...
ζ⋅ε=β    β≤η      ✅       Pushing ε to the list of candidates...
ζ⋅ζ=ζ    ζ≤η      ❌
ζ⋅η=δ    δ≤η      ✅       Pushing η to the list of candidates...

candidates: FiniteTruth[⊥, α, β, δ, ε, η]
```

In [124…

```
candidate = first(candidates)
for new_candidate in candidates[2:length(candidates)]
    if precedes(H9, candidate, new_candidate)    # candidate < new_candida
        candidate = new_candidate
    end
end
result = candidate
```

Out[124…

```
η
```

In [125…

```
H9.implication(ζ, η)
```

Out[125…

```
η
```

Exercise: try to do the same with other values from the H9 algebra, such as:

- ζ and ϵ
- ζ and β
- α and β

And of course, we can also use the `check` ...

In [126…

```
model = TruthDict([p=>α, q=>β])
```

Out[126…

TruthDict with values:

it's mortal! String	**it's alive!** String
β	α

In [127…

```
check(∧(p,q), model, G4)    # return true only if ⊤
```

Out[127…

```
false
```

and the `interpret` !

In [128…

```
interpret(∧(p,q), model, G4)    # min
```

Out[128…

```
α
```

Some other examples...

In [129…

```
check(∨(p,q), model, H4)
```

Out[129… true

In [130…
```
interpret(∨(p,q), model, H4)    # sup
```

Out[130… ⊤

In [131…
```
model = TruthDict([p=>ζ, q=>η])
```

Out[131…
```
TruthDict with values:
┌──────────────┬──────────────┐
│ it's mortal! │ it's alive!  │
│       String │       String │
├──────────────┼──────────────┤
│            η │            ζ │
└──────────────┴──────────────┘
```

In [132…
```
interpret(∧(p,q), model, H9)
```

Out[132… δ

In [133…
```
interpret(→(p,q), model, H9)
```

Out[133… η

Esercise: try the same exercises from before, but this time use the `interpret` function instead.

SoleReasoners.jl (extra)

One classical problem we would like to solve is satisfiability (SAT).

I.e., given a formula in a specific logic, is there a model that satisfies that model?

The literature provides us with many tools to do that, called SAT solvers.

`Sole.jl` provides its own package, called `SoleReasoners.jl`, which works out of the box with `SoleLogic.jl`'s syntax.

In [134…
```
Pkg.add(url="https://github.com/aclai-lab/SoleReasoners.jl#embedding")
```

```
    Updating git-repo `https://github.com/aclai-lab/SoleReasoners.jl#embed
ding`
   Resolving package versions...
  No Changes to `~/logic-and-machine-learning/Project.toml`
  No Changes to `~/logic-and-machine-learning/Manifest.toml`
```

In [135…
```
using SoleReasoners
```

In [136…
```
p, q = Atom.(["p", "q"])
```

Out[136…
```
2-element Vector{Atom{String}}:
 Atom{String}: p
 Atom{String}: q
```

In [137…
```
φ = parseformula("p∧q");
sat(φ)
```
Out[137… true

In [138…
```
φ = parseformula("p∧¬p")
sat(φ)
```
Out[138… false

Similarly, we can ask if a formula is valid (i.e., satisfied by any possible model).

Once again, the literature provides us with many tools, namely automated theorem provers.

`SoleReasoners.jl` uses an analytic tableau technique to find if a formula is satistiable or not: hence, if we want to ask if a formula is valid or not, we just have to negate the input formula and searching for a countermodel (hence, we have to invert the input).

In [139…
```
using SoleReasoners: prove as val    # function isn't exported by default
```
In [140…
```
φ = parseformula("p∧q");
val(φ)
```
Out[140… false

In [141…
```
φ = parseformula("p∨⊤");
val(φ)
```
Out[141… true

`SoleReasoners.jl` offers also support for Modal Logic K; however, it doesn't define an algorithm explicitly, but rather levarage an already existing sat solver (*spartacus*).

In the sake of time, we won't see it here (we would have to install this other third party tool).

On the other hand, it also supports the four multi-modal logic saw above, as well as (finite) fuzzy and many-valued logics!

Well... actually it supports something more general than that... which is a combination of multi-modal and many-valued logics!

Don't stress too much about it: we can still use it for either (crisp) multi-modal or propositional many-valued logic!

In [142…
```
# Let's introduce some aliases in the hope to create less confusion
LTLFPTableau = MVLTLFPTableau
CLTableau = MVCLTableau
HSTableau = MVHSTableau
LRCC8Tableau = MVLRCC8Tableau
```

```
# For many-valued logic (including fuzzy logic), we could use any of the 
MVTableau = MVLTLFPTableau
;
```

Let's start by defining some atoms.

In [143…
```
p, q, r, s = @atoms p q r s
```

Out[143…
```
4-element Vector{Atom{String}}:
 Atom{String}: p
 Atom{String}: q
 Atom{String}: r
 Atom{String}: s
```

Let's see some examples with HS...

First, we need to define the diamond and box operators for the relations we'd like to use.

In [144…
```
diamondA = diamond(HS_A)     # diamond for after (at leat one interval aft
boxA = box(HS_A)             # box for after (all intervals after...)
```

Out[144…
```
BoxRelationalConnective{SoleLogics._IA_A}: [A]
```

Let's try the following formula (for now, we will prefer functional notation).

In [145…
```
φ = ∧(diamondA(p), boxA(→(p, ⊥)))    # this is, of course, a contraddictio
```

Out[145…
```
SyntaxBranch: ⟨A⟩p ∧ [A](p → ⊥)
```

Even if we are just interested in the crisp version of the logic, we have to use a function called `alphasat`, which name will be clear in a minute.

Moreover, we have to specify which kind of tableau we are using, given in the form `LOGICTableau`, and the many-valued algebra we are using; since we are in the crisp case, we will use the `booleanalgebra`.

In [146…
```
using SoleLogics.ManyValuedLogics: booleanalgebra

alphasat(HSTableau, ⊤, φ, booleanalgebra)    # Don't mind the ⊤: we need i
```

Out[146…
```
false
```

We can check that the formula is in fact a contraddiction in the following way:

In [147…
```
alphaval(HSTableau, ⊤, →(φ, ⊥), booleanalgebra)
```

Out[147…
```
true
```

What about many-valued logics?

`alphasat` is in fasct a generalization of the `sat` problem: we are no longer asking if there exists a model so that is satisfies a formula, but rather if there is one so that the valuation of the formula is, at least, some specified value α from the algebra.

That's why we needed to specify $\top$ in the previous example!

Here's an example using Goedel Logic with 3 values...

In [148…
```
using SoleLogics.ManyValuedLogics: G3

φ = →(∨(p, ⊤), α)
```

Out[148…
```
SyntaxBranch: p ∨ ⊤ → α
```

Of course, the $p \vee \top$ subformula is always true in G3.

In [149…
```
alphaval(MVTableau, ⊤, ∨(p, ⊤), G3)
```

Out[149…
```
true
```

So, the previous formula could've been written $\top \to \alpha$; it's just syntax sugar.

On the other hand, $\top \to \alpha$ in G3 equals α.

In [150…
```
G3.implication(⊤, α)
```

Out[150…
```
α
```

That means that `sat` for this formula would return false!

In [151…
```
alphasat(MVTableau, ⊤, φ, G3)
```

Out[151…
```
false
```

But what if we asked that the valuation was at least α?

In [152…
```
alphasat(MVTableau, α, φ, G3)
```

Out[152…
```
true
```

Same goes for `alphaval` (given any model, the valuation of the formula is at least α).

In [153…
```
alphaval(MVTableau, α, φ, G3)    # This was an wasy case: φ doedn't depend
```

Out[153…
```
true
```

Exercise: play with the other logics (LTLFP, CL, LRCC8) and the other algebras (G4, Ł3, Ł4, H4, H9).