Symbolic knowledge manipulation with SoleLogics.jl

Propositional Logic

Formulas & Interpretations

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

```
  Activating project at `~/Desktop/modal-symbolic-learning-course`
    Updating registry at `~/.julia/registries/General`
    Updating git-repo `https://github.com/JuliaRegistries/General.git`
  No Changes to `~/Desktop/modal-symbolic-learning-course/Project.toml`
  No Changes to `~/Desktop/modal-symbolic-learning-course/Manifest.toml`
Status `~/Desktop/modal-symbolic-learning-course/Project.toml`
  [a93c6f00] DataFrames v1.6.1
  [7806a523] DecisionTree v0.12.4
  [7073ff75] IJulia v1.24.2
  [033835bb] JLD2 v0.4.38
⌅ [add582a8] MLJ v0.19.5
  [c6f25543] MLJDecisionTreeInterface v0.4.0
  [e54bda2e] ModalDecisionTrees v0.3.3
  [91a5bcdd] Plots v1.39.0
  [7b3b3b3f] Sole v0.3.1
  [b002da8f] SoleLogics v0.6.12
  [4249d9c7] SoleModels v0.5.3
  [2913bbd2] StatsBase v0.34.2
  [9a3f8284] Random
Info Packages marked with ⌅ have new versions available but compatibility co
nstraints restrict them from upgrading. To see why use `status --outdated`
```

In [43]:
```
using SoleLogics
```

In [46]:
```
p = Atom("it's alive!")
```

Out[46]:
```
Atom{String}: it's alive!
```

In [47]:
```
q = Atom("it's mortal!")
```

Out[47]:
```
Atom{String}: it's mortal!
```

In [48]:
```
φ = p ∧ q
```

Out[48]: SyntaxBranch{NamedConnective{:∧}}: it's alive! ∧ (it's alive! ∧ it's mortal! ∨ it's alive! ∧ it's mortal!)

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

Out[6]: true

In [56]:
```
# φ is the root node of a syntax tree.
# Print the syntax token at the root node
println(token(φ))
# Print the children of the root node
println(children(φ))
```

```
∧
(Atom{String}: it's alive!, SyntaxBranch{NamedConnective{:∨}}: it's alive! ∧ it's mortal! ∨ it's alive! ∧ it's mortal!)
```

In [8]:
```
# Create a method for negating any formula
function negateformula(f::Formula)
    return ¬f
end
```

Out[8]: negateformula (generic function with 1 method)

In [57]:
```
negateformula(φ)
```

Out[57]: SyntaxBranch{NamedConnective{:¬}}: ¬(it's alive! ∧ (it's alive! ∧ it's mortal! ∨ it's alive! ∧ it's mortal!))

In [10]:
```
negateformula(p)
```

Out[10]: SyntaxBranch{NamedConnective{:¬}}: ¬it's alive!

In [11]:
```
# Obtain the string representation of a Formula
syntaxstring(φ)
```

Out[11]: "it's alive! ∧ it's mortal!"

In [69]:
```
# I can also parse Formula's from standard string representations
φ = parseformula("it's alive! ∧ it's mortal!")
```

Out[69]: SyntaxBranch{NamedConnective{:∧}}: it's alive! ∧ it's mortal!

In [13]:
```
function my_own_string_representation(f::SyntaxBranch{NamedConnective{:¬}})
    return "It is not the case that $(my_own_string_representation(first(chi
end
function my_own_string_representation(f::SyntaxBranch{NamedConnective{:∧}})
    subformula1, subformula2 = children(f)
    return "both $(my_own_string_representation(subformula1)) and $(my_own_s
end
function my_own_string_representation(f::SyntaxTree)
    return syntaxstring(f)
end
```

Out[13]: my_own_string_representation (generic function with 3 methods)

In [14]:
```
my_own_string_representation(¬φ)
```

Out[14]: "It is not the case that both it's alive! and it's mortal!"

In [15]:
```
soul = TruthDict([p => true, q => false])
```

Out[15]: TruthDict with values:

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

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

Out[16]: TruthDict with values:

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

In [17]:
```
check(p, soul) # soul is alive
```

Out[17]: true

In [18]:
```
check(φ, soul) # But not both alive and mortal
```

Out[18]: false

In [19]:
```
check(φ, body) # body is both alive and mortal!
```

Out[19]: true

In [20]:
```
# These objects can actually be used as dictionaries from Formula to Truth v
# For example, we both assign ⊤ (top) to the atom "alive"
soul[q], body[q]
```

Out[20]: (⊥, ⊤)

In [64]:
```
body[q]
```

Out[64]: ⊤

In [67]:
```

```

Out[67]: Top

```
In [21]: # What is ⊤?
         ⊤ isa Truth      &&     ⊤ isa Formula

Out[21]: true

In [22]: # By the way, if you find these Unicode characters uncomfortable to work wit
         (TOP, BOT, CONJUNCTION, DISJUNCTION, IMPLICATION)

Out[22]: (⊤, ⊥, ∧, ∨, →)

In [73]: # Actually, indexing (with [·]) can be used to check generic Formula's
         soul[φ]

Out[73]: ⊥

In [70]: # This is syntactic sugar for the *interpretation* algorithm, which is actua
         interpret(φ, soul)

Out[70]: ⊥

In [75]: # In fact, it also works under incomplete information.
         body[φ ∧ Atom("?Unknown property?")]

Out[75]: Atom{String}: ?Unknown property?

In [26]: # Notice how in this example, with an *unknown atom*, it uses the *known* ir
         body[φ ∨ Atom("?Unknown property?")]

Out[26]: ⊤

In [27]: # So ultimately, `check` is just a shortcut for making sure that `interpret`
         check(φ, soul) == istop(interpret(φ, soul))

Out[27]: true

In [28]: # Let's generate random formulas
         treeheight = 3
         @atoms a b
         φ2 = randformula(treeheight, [a,b], SoleLogics.BASE_PROPOSITIONAL_CONNECTIVE

Out[28]: SyntaxBranch{NamedConnective{:∨}}: ¬a ∨ a ∧ b ∨ a ∧ b ∧ (b → b)

In [29]: # Let's control reproducibility, though ;)
         using Random
         rng = Random.MersenneTwister(1)
         φ2 = randformula(rng, treeheight, [a,b], SoleLogics.BASE_PROPOSITIONAL_CONNE

Out[29]: SyntaxBranch{NamedConnective{:¬}}: ¬(b ∧ a ∧ ¬b)

In [30]: normalize(φ2)

Out[30]: SyntaxBranch{NamedConnective{:∨}}: ¬a ∨ b ∨ ¬b
```

Exercise 1:

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 [31]:
```
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[31]: "A2 < 10"

In [32]:
```
using SoleLogics: AbstractAssignment # Abstract type for propositional Inter

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

import SoleLogics: interpret

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

Out[32]: interpret (generic function with 12 methods)

In [33]:
```
rng = Random.MersenneTwister(1)
n_variables = 4

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

Out[33]: TabularInterpretation{Float64}([0.23603334566204692, 0.34651701419196046, 0.3127069683360675, 0.00790928339056074])

In [34]:
```julia
A = Atom.([ConditionOnAttribute(v, t) for v in 1:n_variables for t in 0:0.1:
syntaxstring.(A)
```

Out[34]:
```
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 [35]:
```julia
[interpret(cond, I) for cond in A]
```

Out[35]:
```
44-element Vector{BooleanTruth}:
 ⊥
 ⊥
 ⊥
 ⊤
 ⊤
 ⊤
 ⊤
 ⊤
 ⊤
 ⊤
 ⊤
 ⊥
 ⊥
 ⋮
 ⊤
 ⊥
 ⊤
 ⊤
 ⊤
 ⊤
 ⊤
 ⊤
 ⊤
 ⊤
 ⊤
 ⊤
```

In [36]:
```
rng = Random.MersenneTwister(32)

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

Out[36]:
```
10-element Vector{Pair{String}}:
                 "(¬A2 < 0.4 ∧ (A1 < 0.5 → A4 < 0.8)) → (A4 < 0.4 ∧ A4 < 0.
9 ∧ A2 < 0.9 ∧ A1 < 0.2)" => ⊤
                     "A4 < 0.1 ∨ A2 < 0.6 ∨ A4 < 0.8 ∨ A2 < 0.5 ∨ ¬A4 < 0.8
∧ (A2 < 0.4 → A1 < 0.5)" => ⊤
                                                    "¬(A3 < 0.6 ∧ A4 < 0.9)
∧ ¬(A2 < 0.3 → A1 < 1.0)" => ⊥
                                                 "¬¬A4 < 0.6 ∧ ((¬A1 < 0.0)
→ (A2 < 0.4 ∧ A2 < 0.3))" => ⊥
   "((A4 < 0.4 → A4 < 1.0) ∨ A1 < 0.7 ∨ A2 < 0.7) → ((A3 < 0.3 → A4 < 0.3)
∧ (A2 < 0.2 ∨ A3 < 0.5))" => ⊤
 "((A4 < 0.0 → A2 < 0.3) → (A4 < 0.2 ∧ A3 < 0.9)) ∨ ((A1 < 0.8 → A1 < 0.4)
→ (A2 < 0.7 → A2 < 0.0))" => ⊤
                           "¬A3 < 0.6 ∧ ¬A3 < 1.0 ∧ ((A3 < 0.5 → A3 < 0.5)
→ (A3 < 0.8 ∧ A4 < 0.3))" => ⊥
                                                                 "¬(A3 < 0.6
∧ A1 < 0.1 ∧ ¬A1 < 0.7)" => ⊤
                                                    "¬(A1 < 0.0 ∨ A3 < 0.5)
∧ ¬(A4 < 0.2 ∨ A1 < 0.8)" => ⊥

"¬(¬A1 < 0.5 ∨ ¬A3 < 0.8)" => ⊤
```

Exercise 2

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

Modal Logic

In [37]:
```
# Instantiate a frame with 5 worlds and 5 edges
using Graphs

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

Out[37]:
```
SoleLogics.ExplicitCrispUniModalFrame{SoleLogics.World{Int64}, SimpleDiGrap
h{Int64}} with
- worlds = ["1", "2", "3", "4", "5"]
- accessibles =
        1 -> [2, 3]
        2 -> [4]
        3 -> [4, 5]
        4 -> []
        5 -> []
```

In [38]:
```
# Pick the first world
w1 = worlds[1]

# Enumerate the world that are accessible from the first world
accessibles(fr, w1)
```

Out[38]:
```
2-element Vector{SoleLogics.World{Int64}}:
 SoleLogics.World{Int64}(2)
 SoleLogics.World{Int64}(3)
```

In [39]:
```
# Assign each world a propositional interpretation
valuation = Dict([
        worlds[1] => TruthDict([p => TOP, q => BOT]),
        worlds[2] => TruthDict([p => TOP, q => TOP]),
        worlds[3] => TruthDict([p => TOP, q => BOT]),
        worlds[4] => TruthDict([p => BOT, q => BOT]),
        worlds[5] => TruthDict([p => BOT, q => TOP]),
     ])

# Instantiate a Kripke structure
K = KripkeStructure(fr, valuation)
```

Out[39]:
```
KripkeStructure{SoleLogics.ExplicitCrispUniModalFrame{SoleLogics.World{Int6
4}, SimpleDiGraph{Int64}}, Dict{SoleLogics.World{Int64}, TruthDict{Dict{Ato
m{String}, BooleanTruth}}}} with
- frame = SoleLogics.ExplicitCrispUniModalFrame{SoleLogics.World{Int64}, Si
mpleDiGraph{Int64}} with
- 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! => ⊥])
```

In [40]:
```
# Generate a random modal formula
φmodal = randformula(Random.MersenneTwister(14), 3, [p,q], SoleLogics.BASE_M

println(syntaxstring(φmodal))

# Check the formula on each world of the Kripke structure
[w => check(φmodal, K, w) for w in worlds]
```

```
□(it's alive! → it's mortal!) ∨ ¬(it's alive! → it's alive!)
```

Out[40]:
```
5-element Vector{Pair{SoleLogics.World{Int64}, Bool}}:
 SoleLogics.World{Int64}(1) => 0
 SoleLogics.World{Int64}(2) => 1
 SoleLogics.World{Int64}(3) => 1
 SoleLogics.World{Int64}(4) => 1
 SoleLogics.World{Int64}(5) => 1
```

Exercise 3

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

Exercise 4

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.

Note: Julia currently does not allow redefining structures at the global scope, so the code is wrapped in a module.

In [41]:
```
module exercise4

export ModalInterpretation

using Main: ConditionOnAttribute

using SoleLogics

using SoleLogics: AbstractFrame, World, AbstractKripkeStructure

# TODO:
# struct ModalInterpretation{FR<:AbstractFrame,T<:Real} <: AbstractKripkeStr
#     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,
#     cond = value(a)
#     v = ???
#     return (v < cond.threshold ? TOP : BOT)
# end


end # end module
```

```
UndefVarError: `ModalInterpretation` not defined

Stacktrace:
 [1] top-level scope
   @ In[41]:20
```

In [42]:
```
using .exercise4

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

```julia
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

    println(syntaxstring(φmodal) => ["w$(SoleLogics.name(w))" => check(φmoda
    println()
end
```

```
UndefVarError: `ModalInterpretation` not defined

Stacktrace:
 [1] top-level scope
   @ In[42]:14
```