

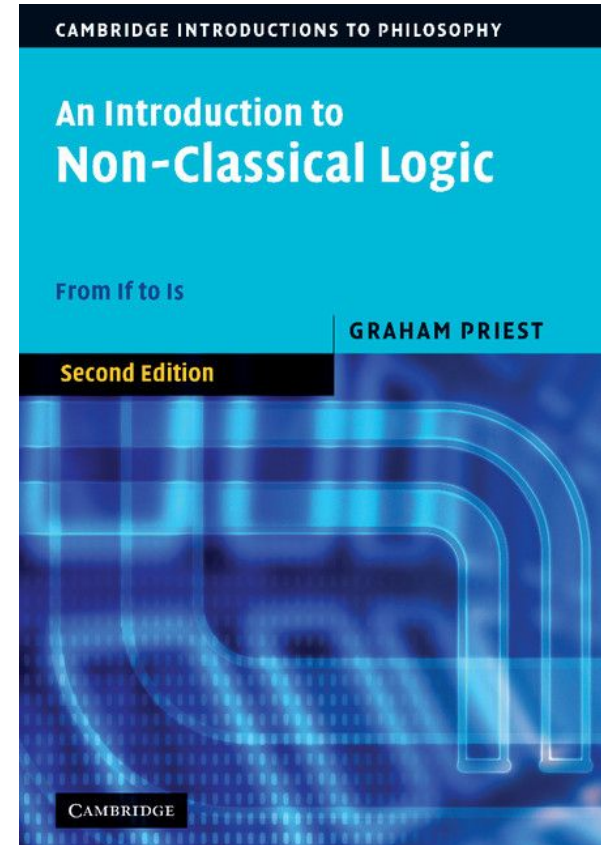
Developing an Automated Proof Calculator for Modal Logic

**Based on Graham Priest - “An Introduction to
Non-Classical Logic. From if to is (second edition)”**

Andrei Dobrescu (andrei.dobrescu@neurony.ro)
and Marian Călborean (mc@filos.ro)
(work in progress)

Tableaux proof system

- A strictly syntactic proof system, just as natural deduction and axiomatic systems.
- In “An Introduction to Non-Classical Logic. From if to is” (2008, right), Graham Priest theorizes tableaux systems for various non-classical logics: modal logics, intuitionistic logic, many-valued logics and their first-order counterparts.
- The software is free and open-source.
 - Available online at: <https://andob.io/incl/>
 - Written in Rust, source code [here](#).



The calculator software

Problem About

Operator notations:

Logic:

Premises:

Conclusion:

PROVED!

Problem Catalog

- Propositional logic
 - DoubleNegation AndAssociativity OrAssociativity AndDistributivity OrDistributivity DeMorgan1 DeMorgan2 1.14.1.a 1.14.1.b 1.14.1.c 1.14.1.d 1.14.1.e 1.14.1.f 1.14.1.g 1.14.1.h 1.14.1.i 1.14.1.j
- First order logic
 - Socrates Buddha Epicurus 12.13.2.a 12.13.2.b 12.13.2.c 12.13.2.d 12.13.3.a 12.13.3.b 12.13.3.c 12.13.3.d 12.13.3.e 12.13.3.f 12.13.5.a.1 12.13.5.a.2 12.13.5.a.3 12.13.5.b.1 12.13.5.b.2 12.13.5.b.3 12.13.5.c.1 12.13.5.c.2 12.13.5.c.3 12.13.5.c.4 12.13.5.c.5 12.13.5.c.6 12.13.5.c.7 12.13.5.c.8 12.13.5.d 12.13.5.e 12.13.5.f 12.13.5.g 12.13.6.a 12.13.6.b 12.13.6.c 12.13.6.d 12.13.6.e 12.13.6.f 12.13.8.a 12.13.8.b 13.10.2.a 13.10.2.b 13.10.2.c 13.10.2.d 13.10.2.e 13.10.2.f 13.10.2.g 13.10.2.h 13.10.2.i 13.10.6.a 13.10.6.b 13.10.6.c 13.10.6.d
- Propositional normal modal logic
 - 2.12.2.a 2.12.2.b 2.12.2.c 2.12.2.d 2.12.2.e 2.12.2.f 2.12.2.g 2.12.2.h 2.12.2.i 2.12.2.j 2.12.2.k 2.12.2.l 2.12.2.m 2.12.2.n 2.12.2.o 2.12.2.p 2.12.2.q 2.12.2.r 2.12.2.s 2.12.2.t 2.12.2.u 2.12.2.v 3.10.3.a 3.10.3.b 3.10.3.c 3.10.3.d 3.10.3.e 3.10.3.f 3.10.4.a 3.10.4.b 3.10.5.a 3.10.5.b 3.10.5.c 3.10.5.d 3.10.6.a 3.10.6.b 3.10.6.c 3.10.6.d
- First order normal modal logic (constant domain)
 - Barcan 14.10.2.a 14.10.2.b 14.10.2.c 14.10.2.d 14.10.2.e 14.10.2.f 14.10.3.a 14.10.3.b 14.10.3.c 14.10.3.d 14.10.4.a 14.10.4.b 14.10.4.c 14.10.4.d 14.10.4.e 14.10.4.f 14.10.4.g 14.10.4.h 14.10.4.i 14.10.4.j 14.10.4.k 14.10.4.l 14.10.4.m 14.10.4.n 14.10.4.o 14.10.4.p 14.10.4.q 14.10.4.r 14.10.4.s 14.10.4.t 14.10.4.u 14.10.4.v 14.10.4.w 14.10.4.x 14.10.4.y 14.10.4.z 14.10.5.a 14.10.5.b 14.10.5.c 14.10.5.d 14.10.5.e 14.10.5.f 14.10.5.g 14.10.5.h 14.10.5.i 14.10.5.j 14.10.5.k 14.10.5.l 14.10.5.m 14.10.5.n 14.10.5.o 14.10.5.p 14.10.5.q 14.10.5.r 14.10.5.s 14.10.5.t 14.10.5.u 14.10.5.v 14.10.5.w 14.10.5.x 14.10.5.y 14.10.5.z 14.10.6.a 14.10.6.b 14.10.6.c 14.10.6.d 14.10.6.e 14.10.6.f 14.10.6.g 14.10.6.h 14.10.6.i 14.10.6.j 14.10.6.k 14.10.6.l 14.10.6.m 14.10.6.n 14.10.6.o 14.10.6.p 14.10.6.q 14.10.6.r 14.10.6.s 14.10.6.t 14.10.6.u 14.10.6.v 14.10.6.w 14.10.6.x 14.10.6.y 14.10.6.z 14.10.7.a 14.10.7.b 14.10.7.c 14.10.7.d 14.10.7.e 14.10.7.f 14.10.7.g 14.10.7.h 14.10.7.i 14.10.7.j 14.10.7.k 14.10.7.l 14.10.7.m 14.10.7.n 14.10.7.o 14.10.7.p 14.10.7.q 14.10.7.r 14.10.7.s 14.10.7.t 14.10.7.u 14.10.7.v 14.10.7.w 14.10.7.x 14.10.7.y 14.10.7.z
- First order normal modal logic (variable domain)
 - Barcan' 15.12.2.a 15.12.2.b 15.12.2.c 15.12.2.d 15.12.2.e 15.12.2.f 15.12.2.g 15.12.2.h 15.12.2.i 15.12.2.j 15.12.2.k 15.12.2.l 15.12.2.m 15.12.2.n 15.12.2.o 15.12.2.p 15.12.2.q 15.12.2.r 15.12.2.s 15.12.2.t 15.12.2.u 15.12.2.v 15.12.2.w 15.12.2.x 15.12.2.y 15.12.2.z 15.12.3.a 15.12.3.b 15.12.3.c 15.12.3.d 15.12.3.e 15.12.3.f 15.12.3.g 15.12.3.h 15.12.3.i 15.12.3.j 15.12.3.k 15.12.3.l 15.12.3.m 15.12.3.n 15.12.3.o 15.12.3.p 15.12.3.q 15.12.3.r 15.12.3.s 15.12.3.t 15.12.3.u 15.12.3.v 15.12.3.w 15.12.3.x 15.12.3.y 15.12.3.z 15.12.4.a 15.12.4.b 15.12.4.c 15.12.4.d 15.12.4.e 15.12.4.f 15.12.4.g 15.12.4.h 15.12.4.i 15.12.4.j 15.12.4.k 15.12.4.l 15.12.4.m 15.12.4.n 15.12.4.o 15.12.4.p 15.12.4.q 15.12.4.r 15.12.4.s 15.12.4.t 15.12.4.u 15.12.4.v 15.12.4.w 15.12.4.x 15.12.4.y 15.12.4.z 15.12.5.a 15.12.5.b 15.12.5.c 15.12.5.d 15.12.5.e 15.12.5.f 15.12.5.g 15.12.5.h 15.12.5.i 15.12.5.j 15.12.5.k 15.12.5.l 15.12.5.m 15.12.5.n 15.12.5.o 15.12.5.p 15.12.5.q 15.12.5.r 15.12.5.s 15.12.5.t 15.12.5.u 15.12.5.v 15.12.5.w 15.12.5.x 15.12.5.y 15.12.5.z

Proof Tree

◇ ¬C w0

A ≡ B w0

w0Rw1
w1pw1

¬C w1

A ≡ B w1

□(A ≡ B) ⊃ C w1

¬□(A ≡ B) w0

◇ ¬(A ≡ B) w0

A w0

¬A w0

w0Rw2
w2pw2

B w0

¬B w0

¬(A ≡ B) w2

¬□(A ≡ B) w1

C w1 X

C w1 X

Modality Graph

Countermodel

w0

w1

w2

w3

w4

Implementation coverage

(Part I of the book: Propositional logics)

	Chapter	Status
1	Classical logic	✓ Propositional logic fully implemented.
2	Basic modal logic	✓ K modal logic fully implemented.
3	Normal modal logics	✓ T,B,S4,S5 modal logics fully implemented. K tense modal logic partially implemented: for temporal convergence rule, multiple graphs per problem are needed, right now there is a single graph per problem.
4	Non-normal modal logics	✓ S0.5,N,S2,S3,S3.5 modal logics fully implemented.
5	Conditional logics	✓ C fully implemented. C+ partially implemented, multiple graphs per problem are needed.
6	Intuitionist logic	✓ Fully implemented.
7	Many-valued logics	✓ Skip, no tableaux on this chapter.
8	First degree entailment	✓ Regular FDE fully implemented. Routley star FDE variant not implemented.
9	Logics with gaps, gluts and worlds	✓ K4,N4,I4,I3,W logics fully implemented.
10	Relevant logics	✗ Skip, this is really difficult to implement.
11	Fuzzy logics	✓ Skip, no tableaux on this chapter.
11a	Many-valued modal logics	✓ Lukasiewicz logic, Kleene logic, Logic of Paradox, RMingle3 logic fully implemented.

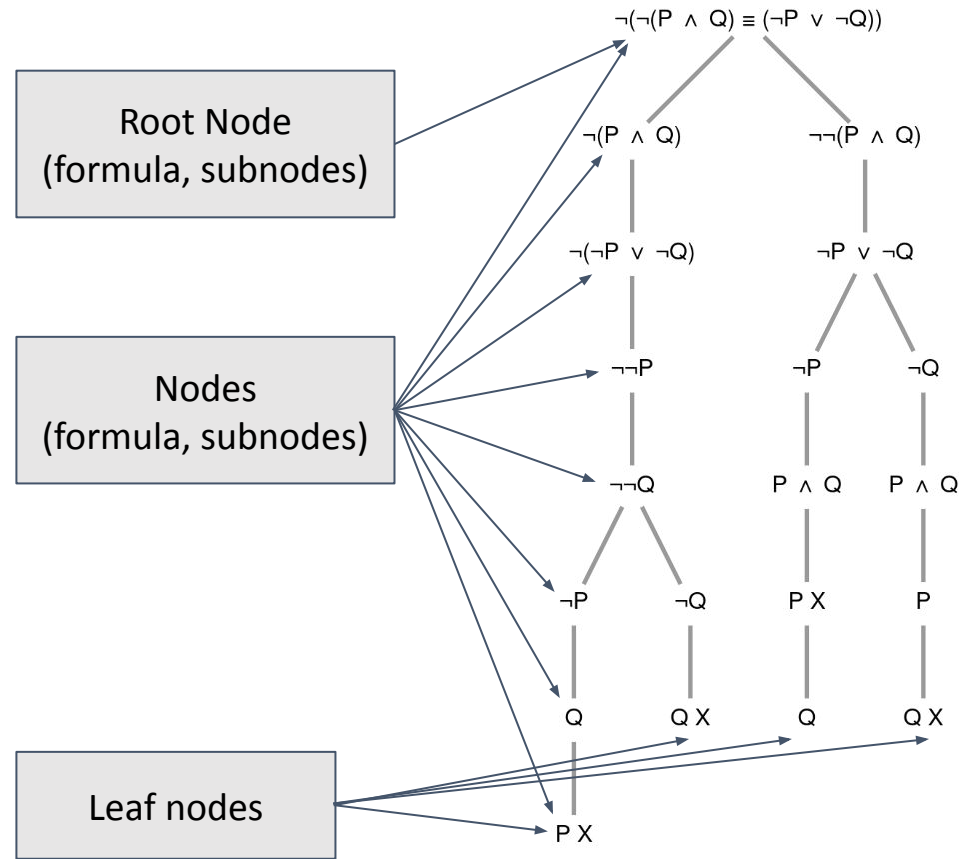
Implementation coverage

(Part II of the book: First-Order counterparts)

	Chapter	Status
12	Classical first-order logic	✓ Fully implemented.
13	Free logics	✓ Implemented only with negativity constraint. Positive free logic is not implemented.
14	Constant domain modal logics	✓ Fully implemented.
15	Variable domain modal logics	✓ Implemented only with negativity constraint.
16	Necessary identity in modal logic	✓ Fully implemented.
17	Contingent identity in modal logic	✓ Fully implemented.
18	Non-normal modal logics	✓ Fully implemented.
19	Conditional logics	✓ C fully implemented. C+ partially implemented.
20	Intuitionist logic	✓ First kind of tableaux implemented. Second kind of tableaux not implemented.
21	Many-valued logics	✓ Fully implemented.
22	First degree entailment	✓ Fully implemented.
23	Logics with gaps, gluts and worlds	✓ Fully implemented.
24	Relevant logics	✗ Skip, this is really difficult to implement.
25	Fuzzy logics	✓ Skip, no tableaux on this chapter.

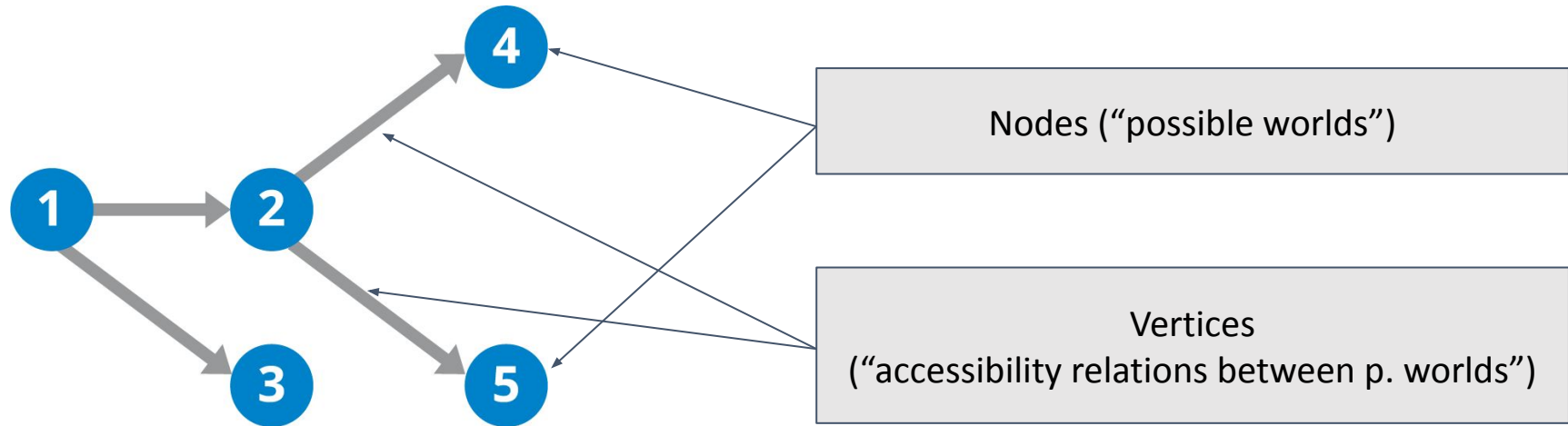
What is a tree?

- A data structure consisting of **nodes** disposed in an arborescent manner.
- Each node has a formula and many **subnodes**.
- Except for the **root node**, each node has a parent node.
- The nodes which have zero subnodes are called **leaf nodes**.
- A path containing all adjacent nodes from the root node to a leaf node is called a **tree branch**.

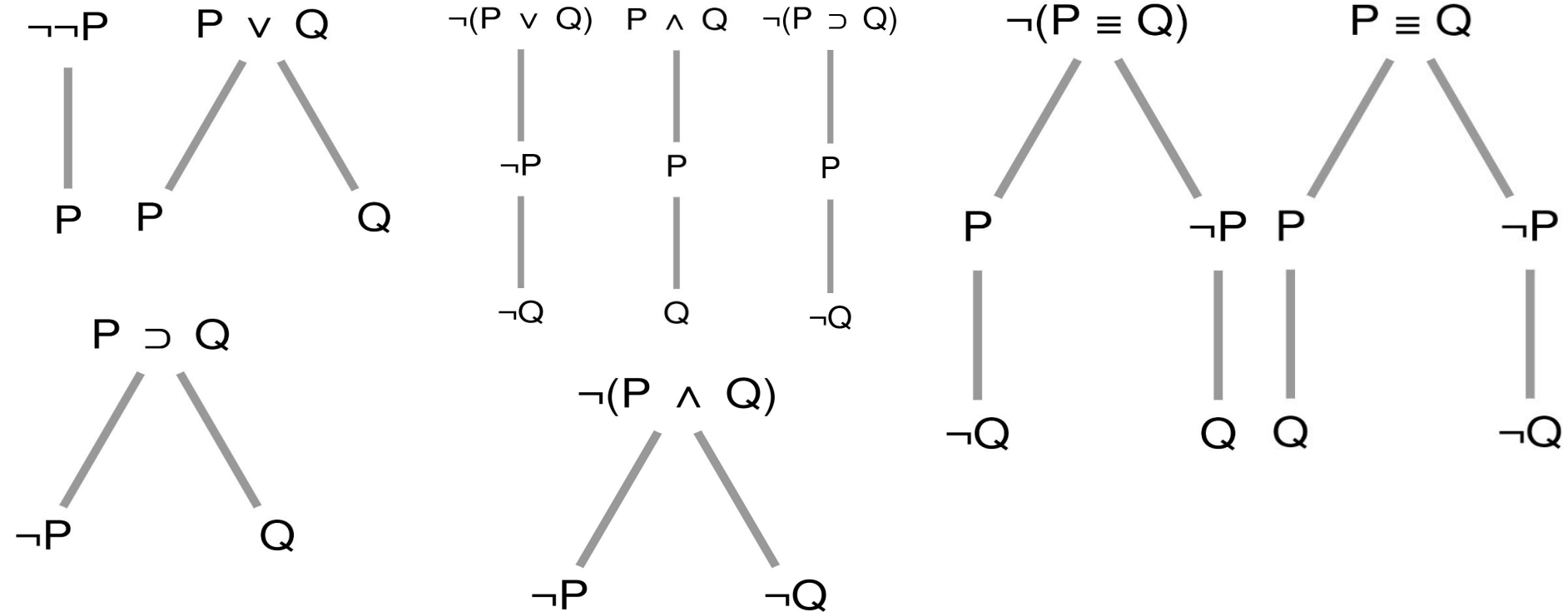


What is a (directed) graph?

- A directed graph is a data structure consisting of a set of **nodes** and a set of **vertices** („arrows” between one node to another).
- The computer science concept of directed graph can be directly mapped to the philosophical concept of a **Kripke model**.



Classical logic rules



Classical logic: example

- Proving DeMorgan rule:
 - $\neg(P \vee Q) \equiv \neg P \wedge \neg Q$
 - ([link](#))

Modal logic rules

 $\Box A, i$
 irj
 \downarrow
 A, j
 $\Diamond A, i$
 \downarrow
 irj
 A, j
 $\neg \Diamond P$
 \downarrow
 $\Box \neg P$
 $\neg \Box P$
 \downarrow
 $\Diamond \neg P$
 $P \rightarrow Q$
 \downarrow
 $\Box (P \supset Q)$
 $\neg (P \rightarrow Q)$
 \downarrow
 $\neg \Box (P \supset Q)$

◇ Possibility rule

◇ ◇ P w0

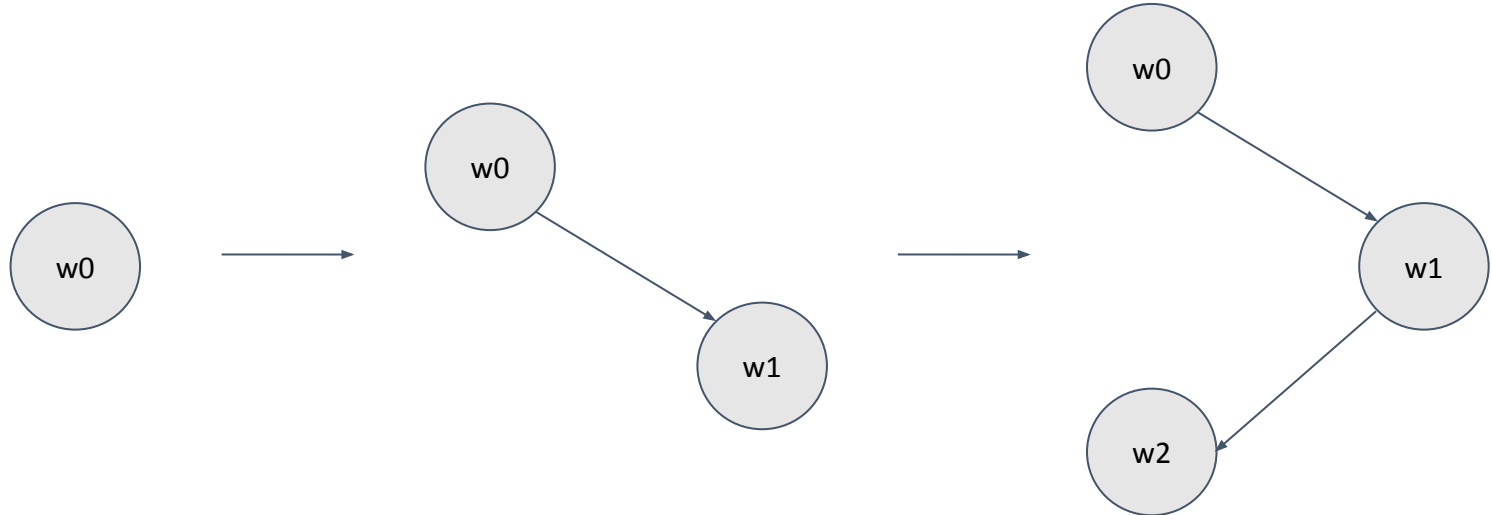
w0Rw1

◇ P w1

w1Rw2

P w2

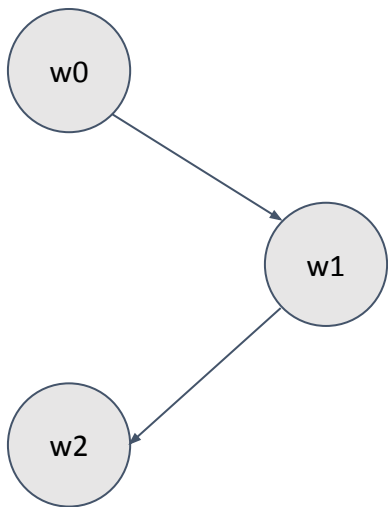
- ◇ rule will spawn a new possible world from the current world.
- ◇ rule will add new nodes and new vertices to the graph.
- ◇ rule will add $w_i R w_j$ nodes to the tree for each new vertex.



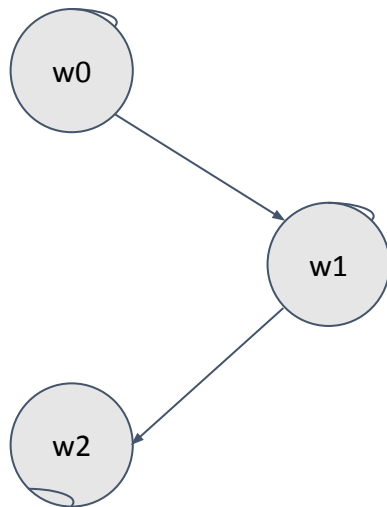
◇ Possibility rule

◇ rule will also add additional graph vertices, as follows:

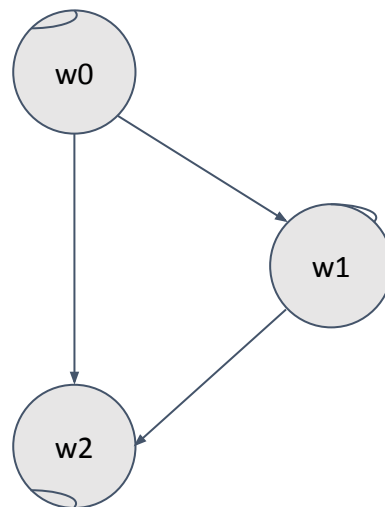
K



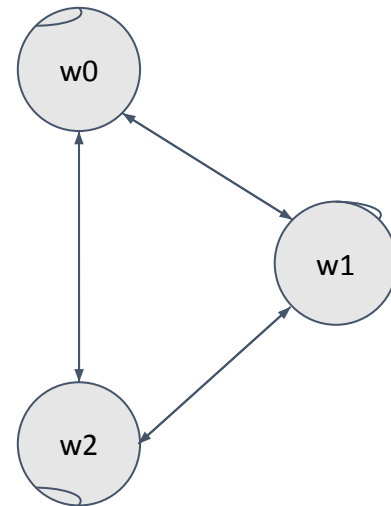
T (reflexive)



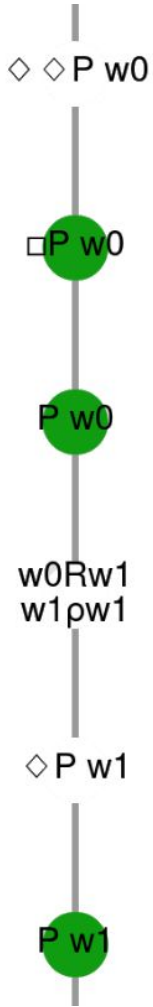
S4 (r+transitive)



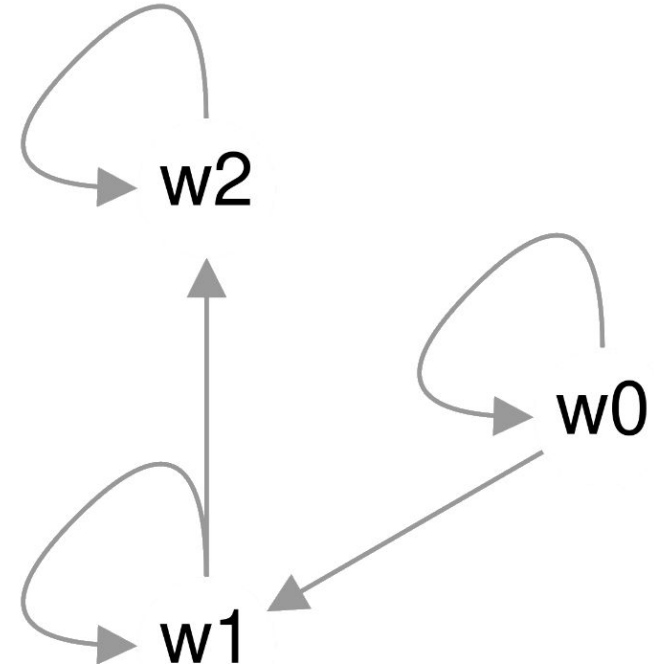
S5 (r+t+symmetric)



□ Necessity rule



- □ rule will iterate all graph vertices that start from the current world:
 - Example: □P w0 is applied here.
 - There are two vertices $w0 \rightarrow w0$ and $w0 \rightarrow w1$. Worlds $w0$ and $w1$ are accessible from world $w0$.
 - Thus the rule will add two nodes to the tree: P w0 and P w1.



Modal logic example

- Proving:
- $\Box A \equiv \neg \Diamond \neg A$
 - ([link](#))

First-order modal logic example

- Proving Barcan formula:
 - $\forall x \Box A[x] \equiv \Box \forall x A[x]$
 - ([link](#))

Philosophical notes

What is a logic?

- In code I define a logic as an object with the following properties:
 - An unique name (eg: PropositionalLogic, S5ModalLogic, FirstOrder+S5ModalLogic, Lukasiewicz+KModalLogic,...)
 - The number of available truth values (2, 3, 4 or ∞)
 - A syntax: a set of symbols available in that logic (eg: $\neg, \wedge, \vee, \rightarrow, \dots$)
 - A set of rules, telling the computer how tree nodes should be generated and how contradictions should be detected.
- This is, of course, not a definition of a logic as a mathematical theory, just an engineered model of the theory.

Why did I use a graph?

- The algorithm, as theorized in the book, does not use a graph data structure, only a tree data structure. In the tree, possible worlds and the relations between them are described via $w_i R w_j$ nodes (meaning: there are two worlds w_i , w_j and a $w_i \rightarrow w_j$ relation between them).
- Since a Kripke model can be built from these nodes, and there is a direct equivalence between a Kripke model and a graph data structure, I chose, based on my programming experience, to use a graph in tandem with the proof tree.
- There are technical reasons for my choice: by using two separate data structures, the code becomes easier to follow, maintain and extend.
- In programming, simplicity over complexity is preferred, even if by simplifying we introduce new constructs.

Thank you!

- **Special thanks**

- Graham Priest, Marian Călborean, Alexandru Dragomir

- **Bibliography**

- Graham Priest - “An Introduction to Non-Classical Logic. From if to is (second edition)” (2008)
- Graham Priest – “Logic: A Very Short Introduction” (2000)
- Melvin Fitting, Richard Mendelsohn – “First-Order Modal Logic” (1998)
- James Storer – “An Introduction to Data Structures and Algorithms” (2002)
- Steve Klabnik, Carol Nichols – “The Rust Programming Language” (2023)

Questions?

Annex: Technical details

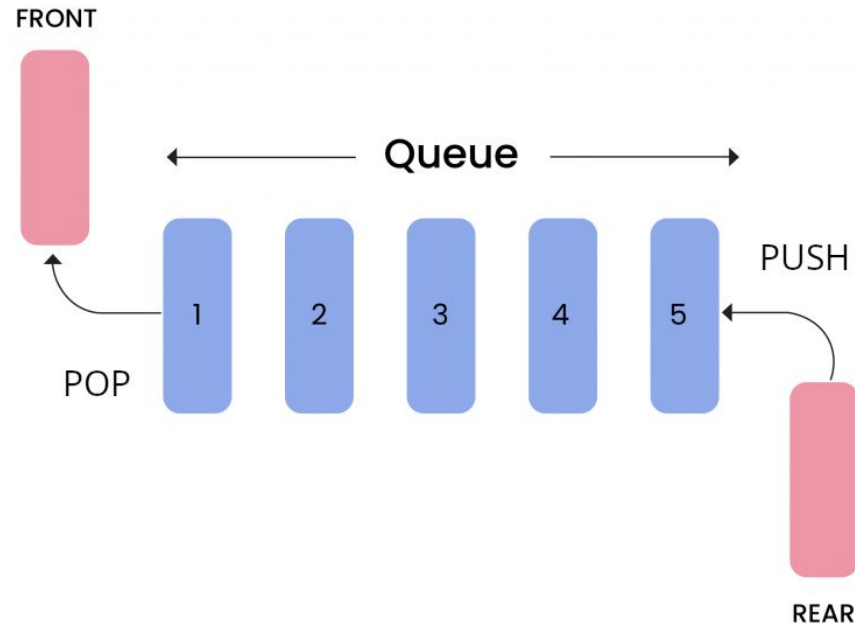
How it works?

The software takes a *Problem* as input and outputs a *ProofTree*.

- The *Problem* consists of a logic, 0..n premises and a conclusion.
- The software can either prove or disprove the problem.
- The proof procedure starts by building the *ProofTree* with vertically disposed sequential nodes. For each premise, there will be a node for that premise. There will be another node with the non-conclusion.
- A *ProofTree* is formed by following specific decomposition rules.
- Problem is proved iff the *ProofTree* has contradiction on all branches (thus the initial assumption about the conclusion is false).
- If the *Problem* is disproved, the program provides a counterexample.
- The software will timeout (neither prove nor disprove) above a limit (if the *ProofTree* reaches 250 nodes).

What is a queue?

- A FIFO (“first in first out”) data structure.
- On the rear of the queue, the program pushes formulas (starting with the premises and the non-conclusion).
- From the front of the queue, the program will sequentially pop and process formulas.
- The program stops when the queue becomes empty (when there are no formulas left to process).

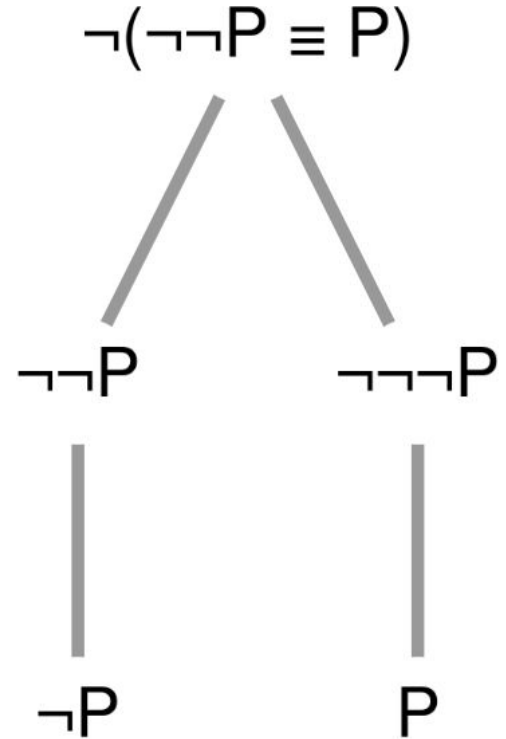


The algorithm in a nutshell - 1/4

- Initial step: given a **Problem** (logic, premises, conclusion):
 - The program creates a **ProofTree** with root node = non-conclusion, and subnodes for each premise.
 - The program creates a queue with these formulas.
- While the queue is not empty:
 - Pop one formula from the queue.
 - Find the specific applicable logic rule for this formula (each logic has different rules, see next slide).
 - Apply the rule and append the result to the ProofTree.
 - Push the resulting formulas to the queue.
 - Check for contradictions in the ProofTree.

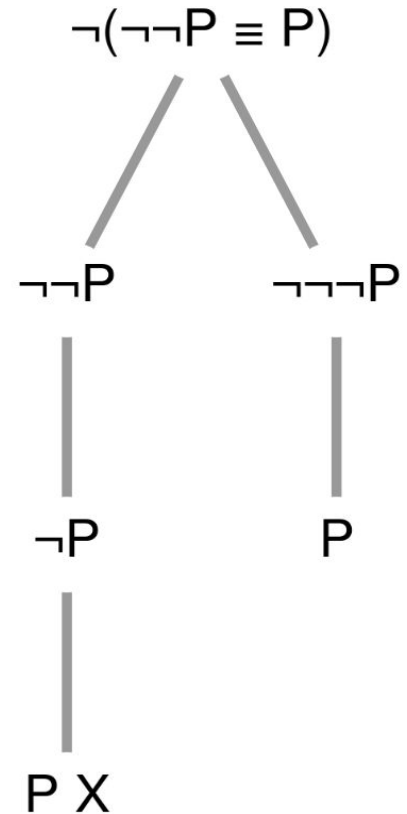
The algorithm in a nutshell - 2/4

- For instance, let's prove that $\neg\neg P \equiv P$
- Initialization:
 - $\text{proofTree} = \{ \text{rootNode: } \neg(\neg\neg P \equiv P) \}$
 - $\text{queue} = \{ \neg(\neg\neg P \equiv P) \}$
- First iteration: since queue is not empty:
 - The $\neg(\neg\neg P \equiv P)$ formula is popped from the queue
 - The queue becomes empty
 - The $\neg \equiv$ rule is applied (result: 4 formulas)
 - The result is appended to the tree
 - Check for contradictions: there are no contradictions
 - All resulting formulas are pushed to the queue
 - The queue becomes $\{ \neg\neg P, \neg P, \neg\neg\neg P, P \}$



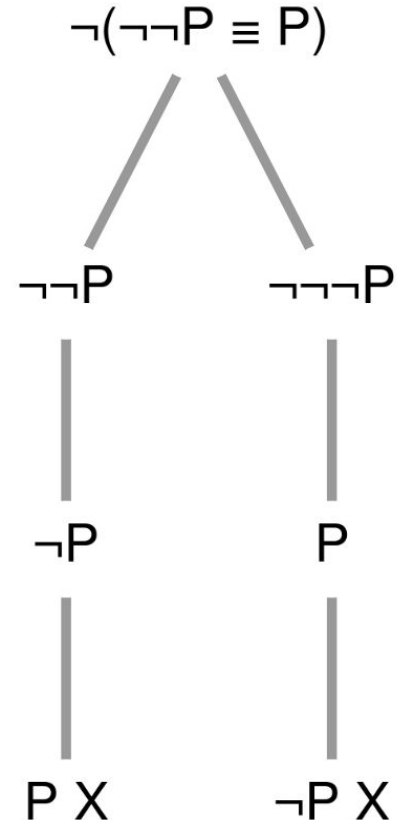
The algorithm in a nutshell - 3/4

- Second iteration
 - The $\neg\neg P$ formula is popped from the queue.
 - The queue becomes $\{ \neg P, \neg\neg\neg P, P \}$
 - The double negation rule is applied
 - Check for contradictions: found a contradiction
 - Add P to queue. The queue becomes $\{ \neg P, \neg\neg\neg P, P, P \}$
- Third iteration
 - The $\neg P$ formula is popped from the queue.
 - The queue becomes $\{ \neg\neg\neg P, P, P \}$
 - There is no rule for $\neg P$. Skip this iteration.



The algorithm in a nutshell - 4/4

- Fourth iteration
 - The $\neg\neg P$ formula is popped from the queue.
 - Queue: $\{ P, P \}$
 - The double negation rule is applied
 - Check for contradictions: found another contradiction
 - Add $\neg P$ to queue. The queue becomes $\{ P, P, \neg P \}$
- 5th / 6th / 7th iteration
 - Pop the queue: $\{ P, P, \neg P \}$.
 - There are no rules to apply. Skip.
- The queue is empty now. Nothing left to do.
 - We have contradiction on all branches
 - Initially we assumed $\neg(\neg\neg P \equiv P)$
 - This can't be right, thus $\neg\neg P \equiv P$ is true.
 - $\neg\neg P \equiv P$ was proved! ■



◇ Possibility rule (details)

◇ rule will add additional graph vertices, as follows:

- K modal logic requires no other changes to the graph.
- T modal logic requires a reflexive graph.
 - The algorithm adds the missing reflexive vertices as follows: for all nodes w_i , a $w_i \rightarrow w_i$ vertex will be added to the graph, if missing.
- S4 modal logic requires a reflexive and transitive graph.
 - The algorithm adds the missing transitive vertices as follows: for all nodes $w_i, w_\square, w_\square$, if the graph contains $w_i \rightarrow w_\square$ and $w_\square \rightarrow w_\square$ vertices, and the graph does not contain a $w_i \rightarrow w_\square$ vertex, then a $w_i \rightarrow w_\square$ vertex will be added to the graph.
- S5 modal logic requires a reflexive, symmetric and transitive graph.
 - The algorithm adds the missing symmetric vertices as follows: for all nodes w_i, w_\square , if the graph contains a $w_i \rightarrow w_\square$ vertex and the graph does not contain a $w_\square \rightarrow w_i$ vertex, then a $w_\square \rightarrow w_i$ vertex will be added to the graph.

□ Necessity rule (details)

□ rule is applied reactively:

- When the program first applies a $\Box P w_i$ rule, it will look at the graph and find all possible worlds w_j that have a $w_i \rightarrow w_j$ vertex. Then for each w_j possible world, it will add a $P w_j$ node to the tree.
- Afterwards, the algorithm can apply $\Diamond P w_i$ rules, which will add more w_k possible worlds and more $w_i \rightarrow w_k$ vertices to the graph.
- The initial $\Box P w_i$ rule gets remembered. Each time a new $w_i \rightarrow w_k$ vertex is added to the graph, $\Box P w_i$ will be reapplied for each newly created worlds. Thus, for each new accessible possible world w_k , the software will add a $P w_k$ node to the tree.

First-Order logic notations

For technical reasons, the software uses non-standard notations:

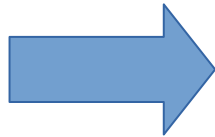
- A predicate with one argument: $P[x]$ (instead of Px)
- A predicate with 2 arguments: $P[x,y]$ (instead of Pxy)
- A predicate with n arguments: $P[x_1,x_2,\dots,x_n]$ (instead of $Px_1x_2\dots x_n$)
- The software categorizes arguments as follows:
 - Free variables (in $P[a] \wedge \exists x Q[x]$, a is a free variable).
 - Binding variables (in $P[a] \wedge \exists x Q[x]$, x is a binding variable).
 - Instantiated objects (in $P[a] \wedge Q[b:x]$, $b:x$ is an instantiated object).
 - $b:x$ notation means „an object named b of type x ”

\exists Existence rule

- The \exists rule will transform binding variables into instantiated objects.
- The rule will generate unique names, considering already existing names on the tree branch.
- Objects are uniquely identified by their names.
- The rule will also attach a type on each object.
Type's name = the name of the original variable.

• Binding variables	Instantiated objects
---------------------	----------------------

• x	a:x
• y	b:y
• z	c:z



$\exists x(\exists y(\exists z(P[x,y,z]))) w_0$

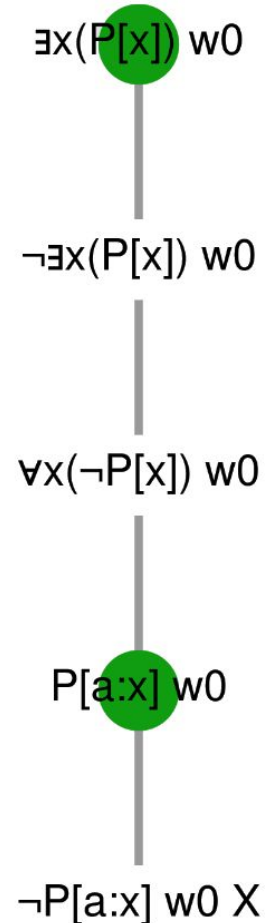
$\exists y(\exists z(P[a:x,y,z])) w_0$

$\exists z(P[a:x,b:y,z]) w_0$

$P[a:x,b:y,c:z] w_0$

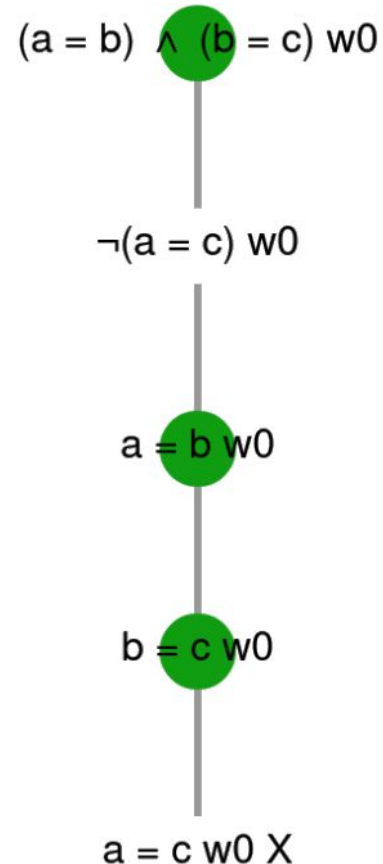
\forall For All rule

- The $\forall x_i$ rule will find on tree branch all instantiated objects of type x_i .
- For each object, the rule will add to the tree a node, replacing x_i with the object.
- For instance, in this example:
 - $\exists x(P[x])$ instantiates x as $a:x$ and adds a $P[a:x]$ node to the tree branch.
 - $\forall x(\neg P[x])$ finds one object of type x (the previously instantiated $a:x$) and adds one node ($\neg P[a:x]$) to the tree branch.



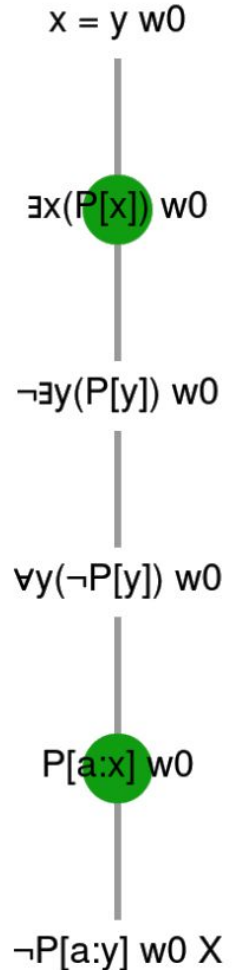
Equality and Contradiction

- The algorithm supports equalities between a variable / object and another variable / object.
 - Transitive equalities are automatically generated: for each $\langle o_1=o_2, o_2=o_3 \rangle$ equality pair, an $o_1=o_3$ equality node will be added, if not already present on branch.
- In FOL, the algorithm detects a contradiction iff:
 - There are two nodes $P[a:x_1, b:x_2, \dots]$ and
 - $\neg P[a:x_1, b:x_2, \dots]$ (same arguments) on the branch.
 - Or there are two nodes $a:x_1 = b:x_2$ and
 - $\neg(a:x_1 = b:x_2)$ (same objects) on the branch.



Equality and \forall For All rule

- The $\forall x_i$ rule will find on tree branch all instantiated objects of type x_i and any other y_i type equivalent to x_i ($x_i = y_i$).
- For each object, the rule will add a node to the tree branch.
- For instance, in this example:
 - $\exists x(P[x])$ instantiates x as $a:x$ and adds a $P[a:x]$ node.
 - $\forall y(\neg P[y])$ finds no objects of type y , has nothing to add.
 - The type y has an equivalent type x ($x = y$).
 - $\forall y(\neg P[y])$ finds one object of equivalent type x (the previously instantiated $a:x$) and adds a $\neg P[x]$ node.



More links

The software is available at: andob.io/incl

Source code (Rust): [here](#)

An initial, beta version: filos.ro/ls/inclcalculator

Source code (Kotlin): [here](#)

A presentation of the beta version: [here](#)