Course Website:

https://kmicinski.com/automated-reasoning

Modern Symbolic AI & Automated Reasoning

Instructor: Kris Micinski

Questions to start us off:

- What **is** reasoning?
  - How can we rigorously define it
  - Can we systematically understand, represent, and implement reasoning procedures?

Example Narrative

Handsome is a dog
Handsome lives in Syracuse
All dogs which live in Syracuse must be vaccinated
Therefore, Handsome must be vaccinated

How can we formalize this?

Handsome is a dog

`dog(andy).`

Handsome lives in Syracuse

`lives(andy, syracuse).`

All dogs which live in Syracuse must be vaccinated

$\forall x.\ \text{dog}(x) \wedge \text{lives}(x, \text{syracuse}) \implies \text{requires\_vaccination}(x)$

Therefore, Handsome must be vaccinated:

Instantiate the quantified formula with `handsome`:

$\texttt{dog(andy)} \wedge \texttt{lives(andy, syracuse)} \implies \texttt{requires\_vaccination(x)}$

Both of the antecedents (things before arrow) are true, thus:

`requires_vaccination(andy)`

Question: how could a **computer** represent this proof?

First, we need to ask ourselves: what's the **logic**?
Broadly, logics formalize (i.e., specify algebraic representation of) formulas, truths, and proofs.

- Our last example used **first-order** logic (FOL) over *atoms*
  - FOL allows universal (∀) and existential (∃) quantifiers
  - Can also include function symbols: $\forall x.\ x > 1 \implies x > 0$
- But also **propositional** logic (no quantifiers)
- Even things like **temporal** logics, which include quantifiers to say things like "in the **future**, the stock price will be higher than it is now."

Now that we picked first-order logic w/ quantifiers, we need to understand how to represent a **proof** of our supposition

For example, we might write a list of valid statements, each either (a) assumptions or (b) statements following from all higher-up statements:

```
(A0) dog(andy).
(A1) lives(andy,syracuse).
(A2) ∀x. dog(x) ∧ lives(x,syracuse)
         ⟹ requires_vaccination(x)
(Instantiate A2, [x ↦ handsome])
dog(andy) ∧ lives(andy,syracuse)
         ⟹ requires_vaccination(andy)
(⟹, A0 A1) requires_vaccination(andy)
```

This style of proof uses a **sequent calculus** formulation
Each line follows from (is conditional upon) *all previous lines*
There are a variety of sequent-style calculi for various logics

```
(A0) dog(andy).
(A1) lives(andy,syracuse).
(A2) ∀x. dog(x) ∧ lives(x,syracuse)
         ⟹ requires_vaccination(x)
(Instantiate A2, [x ↦ andy])
dog(andy) ∧ lives(andy,syracuse)
         ⟹ requires_vaccination(andy)
(⟹, A0 A1) requires_vaccination(andy)
```

The precise formalization of this matters a lot, in terms of reasoning about expressivity, correctness, and completeness (can everything true be proven?) for a given logical system.

In this class, we will detail these philosophical issues, but largely in the context of understanding their impacts on building programs which perform automated reasoning.

E.g., propositional logic is easy (enumerable), but first-order logic (quantifier instantiation) is harder and, in general, requires symbolic search.

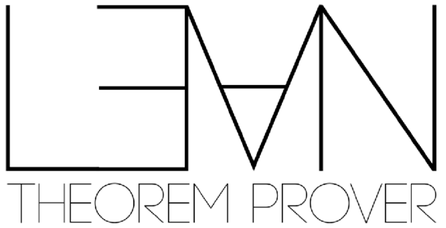
We will ask questions such as:

- How to represent (proofs of) knowledge symbolically?
- How can we build proof checkers, which increase our confidence the proof (system) is meaningful?
- How do you efficiently search for proofs of true statements (or refutations of false statements)?

We will also cover rigorous formal systems necessary to understand these, to the degree necessary to understand the correct design of automated reasoning systems.

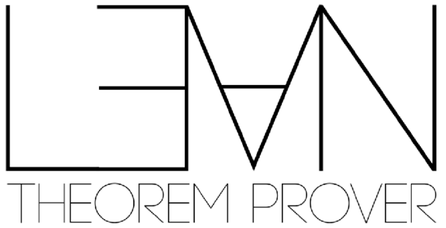
Some Tools we will cover

P1 — SAT Solvers (MiniSat)
P2 — Query languages and Datalog (e.g., Soufflé)
P3 — Constraint solvers and Satisfiability-Modulo Theory Solvers (e.g., Z3, CVC5, etc…)
P4 — Interactive Theorem Provers (Lean)



Projects (in Python)

P1 — Reachability-Based Verification
P2 — Bounded Model Checking w/ SAT
P3 — Symbolic execution with Z3
P4 — Interactive Theorem Provers (Lean)



We use writing to help ourselves structure our thoughts—revising, editing, restarting along the way

This class examines the process of writing and understand programs using a *systematic, iterative approach*

Want to learn “how to think” about programming

Propositional Logic and SAT

CIS700 — Fall 2023

Kris Micinski

Today we'll look at our first logic: propositional logic

Propositional logic consists of formulas built via connectives applied to **atomic propositions**

The following are propositional formulas:
P ("P holds"), every atomic proposition is trivially a formula
$P \wedge (Q \vee \neg P)$ ("P holds and (Q or P) also holds")
$\neg P \wedge Q \implies Q \wedge \neg P$ ("Not P and Q implies Q and not P")
True ($\top$) and False ($\bot$), but these symbols have many meanings

Let’s consider a universe of four propositional variables:
DoorOpen, DoorClosed, MachineOn, MachineOff

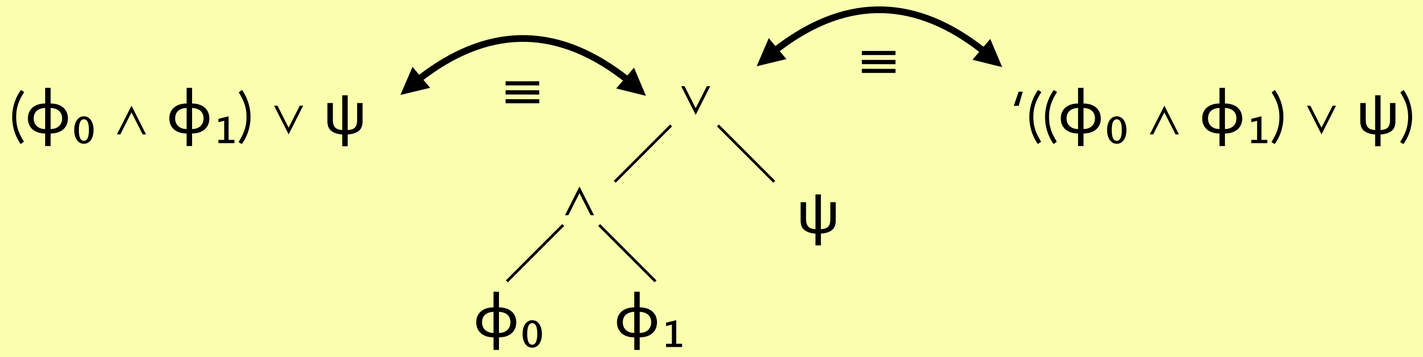
How would you express the following:
“The machine may not be both on and off.”
(Notice that this is xor, even though we excluded it…)

“If the door is open, the machine may not be on.”

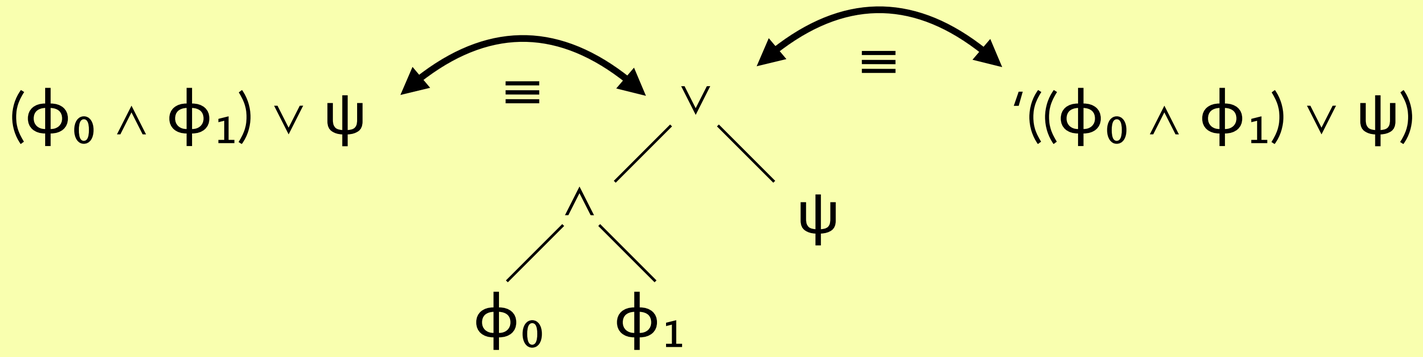
Let’s consider a universe of four propositional variables:
DoorOpen, DoorClosed, MachineOn, MachineOff

How would you express the following:
“The machine may not be both on and off.”
(Notice that this is xor, even though we excluded it…)
$\neg(\text{MachineOn} \wedge \text{MachineOff})$
“If the door is open, the machine may not be on.”
$\text{Open} \Rightarrow \neg\text{MachineOn}$

Propositional formulas are (structurally) recursive structures. All formulas have an implicit recursive structure with constants / propositional variables at their leaves



Scheme's **S-expressions** (structured expressions) systematize this notion. S-expressions are symbolic representations, implemented under the hood via pointers to subtrees



We formalize our expressions as a Racket **predicate**

```
(define (prop-formula? ϕ)
  (match ϕ
    [(? symbol? x) #t]
    ['T #t] ;; for "true"
    ['F #t] ;; for "false"
    [`(¬ ,ϕ) #t]
    [`(,ϕ0 ∧ ,ϕ1) #t]
    [`(,ϕ0 ∨ ,ϕ1) #t]
    [`(,ϕ0 ⇒ ,ϕ1) #t]
    [`(,ϕ0 ⇔ ,ϕ1) #t]
    [_ #f]))
```

We formalize our expressions as a Racket **predicate**

```
(define (prop-formula? ϕ)
  (match ϕ
    [(? symbol? x) #t]
    ['T #t] ;; for "true"
    ['F #t] ;; for "false"
    [`(¬ ,ϕ) #t]
    [`(,ϕ0 ∧ ,ϕ1) #t]
    [`(,ϕ0 ∨ ,ϕ1) #t]
    [`(,ϕ0 ⇒ ,ϕ1) #t]
    [`(,ϕ0 ⇔ ,ϕ1) #t]
    [_ #f]))
```

```
;; Test…
(prop-formula?
  '(ϕ0 ∧ ϕ1) ∧ ψ))
;; #t
```

Interpretations

A formula is just a *statement*. To speak of a statement's veracity, we need to rigorously define the notion of a "true" statement.

There are differing perspectives on this:

- A statement is true precisely when I can materialize a symbolic proof for it
  - This is the **constructive** view, every statement demands *evidence* represented as data
- Everything is either true or false, the **classical** view
  - $\neg\neg P \implies P$ (**excluded middle**)

We will start by looking at a classical, model-theoretic interpretation in which interpretations are mappings of variables to booleans.

Later, we will consider the importance of *proof theory*, which deals with proofs as materialized objects which may be used to formally derive knowledge.

These two perspectives are different sides of the same coin, but it is important to be mindful of their differences

Model Theoretic Interpretations

Propositional logic has a simple notion for classical interpretations: sets. If there are a finite number of atoms under consideration (say $\mathscr{A}$), the finite sets are enumerable and every subset I of $\mathscr{P}(\mathscr{A})$ forms a partition, such that atoms in I are considered "true" and atoms not in I are "false."

Example: $\mathscr{A}$ = {On, Off}, $\mathscr{P}(\mathscr{A})$ = {{}, {On}, {Off}, {On, Off}}
Each set in the power set is an *interpretation*
We apply interpretations to formulas to determine their truth

From Interpretations to Valuaations

Given an interpretation (i.e., set of atoms which are to be construed as “true”), we can recursively define veracity

Exercise:

Say I = {On, Off}. What should the truth value (“true” or “false”) be for each of these formulas ϕ when…

ϕ is “On”

ϕ is “On $\wedge$ ¬Off”

ϕ is “On $\Longrightarrow$ Off”

ϕ is “On $\wedge$ (¬On $\vee$ Off)”

Interpretations in code

In Scheme (Racket), we can represent an interpretation as a dictionary (hash):

```
(define (interpretation? I)
  (and (hash? I)
       (andmap symbol? (hash-keys I))
       (andmap boolean? (hash-values I))))
```

These hashes are implemented efficiently via a data structure which allows *persistent* ~O(1) lookup / insertion

This is a variation of the **mutable** hash tables generally shown in intro DS classes, consider looking up HAMT

These hashes are implemented efficiently via a data structure which allows *persistent* ~O(1) lookup / insertion

This is a variation of the **mutable** hash tables generally shown in intro DS classes, consider looking up HAMT

```
(hash-ref (hash-set (hash 'x 3) 'x 5) 'x)
```

We can now define `(interp-formula ϕ I)`, i.e., ⟦ϕ⟧ which considers several different cases based on the structure of ϕ:

- If ϕ is literally true or false, the answer is #t / #f (Racket's "true")
- If ϕ has the form $\phi_0 \oplus \phi_1$, then ⟦ϕ⟧ is `(⊕ ⟦ϕ₀⟧ ⟦ϕ₁⟧)`
  - Where we assume there is a Racket version of ⊕
- If ϕ has the form ¬ϕ then ⟦ϕ⟧ is `(not ⟦ϕ⟧)`

When the interpretation I results in ϕ being true, we say that I **satisfies** ϕ, often written using the notation $I \models \phi$.

In this case, we will call I a "model" of the propositional statement ϕ.

Propositional logic is *decidable*: the set of possible interpretations is finite (assuming formula size is finite) and you can check each interpretation

A formula is **valid** if it is true in *every* interpretation: $\models \phi$, notice that there is nothing to the left of $\models$, suggesting that *every* I will suffice to satisfy ϕ

These statements are called *tautologies*. Which of the following are tautologies?

(I) $\neg P \vee P \vee (\neg P \wedge P)$, (II) $P \implies P \wedge Q$, (III) $P \wedge Q \implies P$

Proof Theory

Logics are described in different ways. We have seen a model theoretic description of propositional logic, which appeals to a semantics (formalized as interpretations).

By contrast, **proof theory** describes *syntactic* objects (proofs) which represent valid *derivations* of new knowledge from old knowledge

We will study several proof systems throughout the course: natural deduction, resolution, sequent calculus, and analytic tableaux

Normal Forms

Working with arbitrarily-structured formulas means that we need to deal with a bunch of different specific forms ($\vee$, $\wedge$, $\Longrightarrow$, $\Leftrightarrow$, …)

This is messy, since there is a much smaller basis. It is possible to encode all formulas in terms of just $\neg$ and $\wedge$, for example.

There are a variety of **normal forms** for propositional logic, i.e., syntactic restrictions on formulas which do not inhibit expressivity

There are **conversion** algorithms into these normal forms

NNF, CNF, and DNF

Negation Normal Form (NNF): All negations have been pushed as far down as possible. ¬(P ∧ Q) ❌, (¬P ∨ ¬Q) ✅

Conversion algorithm: first expand P⟹Q into ¬P ∨ Q (etc…), next repeatedly apply De-Morgan's laws and cancel ¬¬P to P

Conjunctive Normal Form (CNF): A big conjunction of disjunctions ("clauses"): (A ∨ ¬B ∨ C) ∧ (B ∨ ¬C) ∧ (¬A ∨ ¬B)

To obtain: Start with DNF, then distribute over ∧

Disjunctive Normal Form (DNF): A big disjunction of conjunctions: (A ∧ B ∧ C) ∨ (A ∧ ¬B ∧ ¬C) ∨ ...

Let's convert the following to NNF:

$(A \wedge (B \Leftrightarrow A)) \Longrightarrow B$

$\neg(A \wedge (B \Leftrightarrow A)) \vee B$ (encoding $\Longrightarrow$)

$\neg(A \wedge ((B \Longrightarrow A) \wedge (A \Longrightarrow B))) \vee B$, (encoding $\Leftrightarrow$)

$\underline{\neg(A \wedge ((\neg B \vee A) \wedge (\neg A \vee B)))} \vee B$, (encoding $\Longrightarrow$) Now De Morgan's

$\neg A \vee \underline{\neg((\neg B \vee A) \wedge (\neg A \vee B)))} \vee B$, continue De Morgan's

$\neg A \vee (\underline{\neg(\neg B \vee A)} \vee \underline{\neg(\neg A \vee B)})) \vee B$, and continue…

$\neg A \vee ((\underline{\neg\neg B} \wedge \neg A) \vee (\underline{\neg\neg A} \wedge \neg B))) \vee B$, now cancel $\neg\neg$

$\neg A \vee (B \wedge \neg A) \vee (A \wedge \neg B) \vee B$ (NNF, also in DNF)

Starting from here, how do we get to CNF? Naively: use distributivity,

$A \vee (B \wedge \neg A) \vee (A \wedge \neg B) \vee B$

Group everything as binary so that we can distribute:

$(B \wedge \neg A) \vee (A \vee ((A \wedge \neg B) \vee B)))$

Exercise: use distributivity, cancel double negation, to achieve CNF

$(B \vee (A \vee ((A \wedge \neg B) \vee B))) \wedge (\neg A \vee (A \vee ((A \wedge \neg B) \vee B))$, and then…

$(B \vee (A \vee ((A \vee B) \wedge (\neg B \vee B))) \wedge (\neg A \vee (A \vee ((A \wedge \neg B) \vee B))$,

$(B \vee (A \vee ((A \vee B) \wedge (\neg B \vee B))) \wedge (\neg A \vee (A \vee ((A \vee B) \wedge (\neg B \vee B))))$,

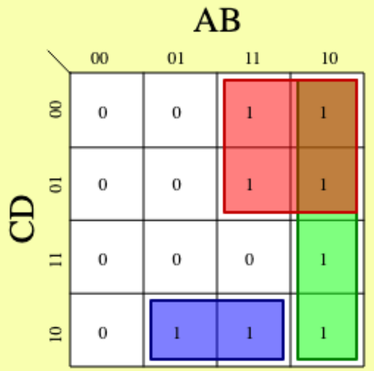
…

$(A \vee B)$

CNF vs. DNF, Digital Logic

In general, we will work with CNF, because it allows a more "dense" representation of formulas. Translation into DNF often results in large (super linear) encoding overhead. Hence, modern solvers often consume input in CNF format.

You may recall this material from a digital logic class—there is serious overlap with minterms/ maxterms/karnaugh maps, worth looking into



Tseitin’s transformation

The distributivity law induces super-linear encoding blowup for formulas—this leads to slowdown of tools which check satisfi ability / validity of these formulas

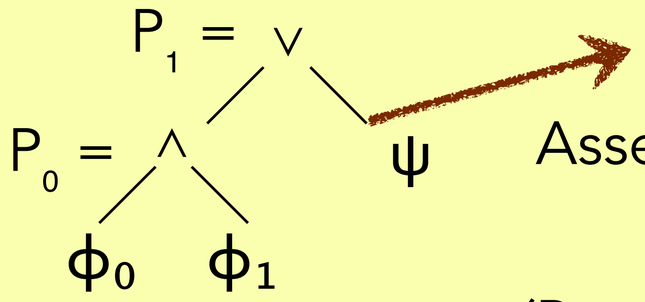
We often want a better transformation from formulas into CNF

A popular transformation is Tseitin’s transformation. Intuitively, it assigns each sub-formula a new propositional atom and asserts a suitable bi-implication

Tseitin's transformation

- Assign non-literal formulas a new variable
- Add definitions for each, recursively

$(\phi_0 \wedge \phi_1) \vee \psi$



$P_1 \Leftrightarrow P_0 \vee \psi$

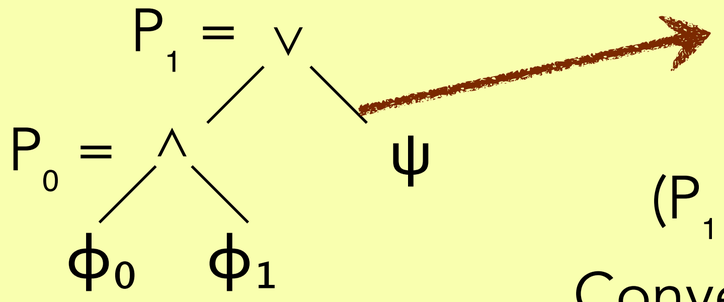
$P_0 \Leftrightarrow \phi_0 \wedge \phi_1$

Assert each definition, plus P_1, via a big conjunction:

$(P_1 \Leftrightarrow P_0 \vee \psi) \wedge (P_0 \Leftrightarrow \phi_0 \wedge \phi_1) \wedge P_1$

Tseitin's transformation

$(\phi_0 \wedge \phi_1) \vee \psi$



$P_1 \Leftrightarrow P_0 \vee \psi$

$P_0 \Leftrightarrow \phi_0 \wedge \phi_1$

Which forms…

$(P_1 \Leftrightarrow P_0 \vee \psi) \wedge (P_0 \Leftrightarrow \phi_0 \wedge \phi_1) \wedge P_1$

Convert each definition to CNF:

The first … $(\neg P_1 \vee P_0 \vee \Psi) \wedge (\neg P_0 \vee P_1) \wedge (\neg \Psi \vee P_1)$

Then… $(\neg P_0 \vee \phi_0) \wedge (\neg P_0 \vee \phi_1) \wedge (\neg \phi_0 \vee P_0) \wedge (\neg \phi_1 \vee P_0)$

Why Tseitin's transformation?

Tseitin's transformation converts to CNF without the super linear blowup of naive distributivity

Output is a set of definitions (defined via bi-implication), where the set size is linear in the size of the input formula (one new var for each node in the formula)

Each of these definitions is of constant depth (either an atom or an application of a connective to atoms; then apply distributivity to obtain CNF (constant factor blowup)

SAT Solving

Checking whether a formula is satisfiable is called SAT(isfiability) solving. Propositional logic is decidable, via truth tables—but what is the complexity of checking via truth tables? Answer: O(2^n)

Does a better solution exist? In practice yes, but only for formulas which obey “typical” structure

Validity solving via SAT?

Notice that whenever ϕ is a validity (i.e., $\models \phi$), $\neg\phi$ is UNSAT

To see why: consider $\neg\phi$ is SAT, then it has some model I such that $I \models \neg\phi$. By assumption ϕ is a validity, so $I \models \phi$ as well. This gives us a contradiction, since $I \models \neg\phi \wedge \phi$

k-SAT

To simplify the core machinery of SAT solving, most solvers operate over CNF, a SAT instance given in CNF is k-SAT if the largest clause consists of k literals

Popular instances include 2-SAT and 3-SAT
2-SAT: $(P \vee \neg Q) \wedge (\neg P \vee Q) \wedge (\neg Q \vee \neg P)$
3-SAT: $(\neg R \vee P \vee \neg Q) \wedge (\neg P \vee Q \vee R) \wedge \ldots$

2-SAT is polynomial time (complete for NL)

2-SAT is a restricted form of k-SAT. Solving 2-SAT can be done in polynomial time and 2-SAT is complete for the class NL (Nondeterministic Logarithm, i.e., Turing machine w/ log memory space)

Algorithm: Start by building a graph of each literal in the program, then for each clause $\phi_0 \vee \phi_1$ add an edge between $\neg\phi_0$ and ϕ_1 and $\neg\phi_1$ and ϕ_0, obtaining a graph G

Last, calculate strongly-connected components of G and check if there is any P such that P and ¬ are in the same SCC

2-SAT Algorithm: Intuition

$$(P \vee \neg Q) \wedge (\neg P \vee Q) \wedge (\neg Q \vee \neg P)$$

Algorithm intuition: every clause has the form $\phi_0 \vee \phi_1$: if $\neg\phi_1$ holds, then the <u>only thing</u> that can make the clause true (and it *must* be true in any satisfying assignment!) will be ϕ_0. Thus, $\neg\phi_1$ "forces" ϕ_0 (unit propagation).

Knowing this, we (a) assemble a "forced-implication" graph and then (b) find its strongly-connected components, finally (c) checking if there are any literals P such that P and ¬P are in the same SCC

Solving 2-SAT via forced implication

$(P \vee \neg Q) \wedge (\neg P \vee Q) \wedge (\neg Q \vee \neg P)$

First, build a graph of literals…

P $\neg P$

Q $\neg Q$

Building forced implication graph

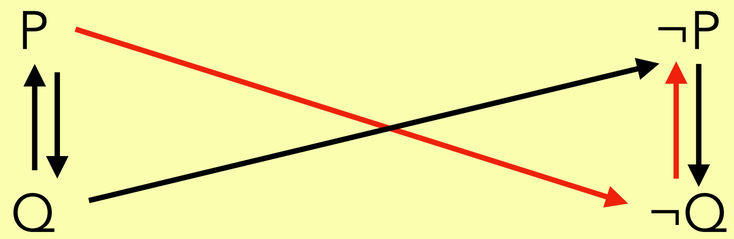
$\underline{(P \vee \neg Q)} \wedge (\neg P \vee Q) \wedge (\neg Q \vee \neg P)$

Now, add edges for each clause—start w/ left one…
(Stay in NNF: cancel $\neg\neg Q$ to Q!)

Calculate SCCs and check

$(P \vee \neg Q) \wedge (\neg P \vee Q) \wedge (\neg Q \vee \neg P)$

Now, add edges for each clause—start w/ left one…
(Stay in NNF: cancel ¬¬Q to Q!)



This is one big SCC—thus, the formula is unsatisfiable

(E.g., see the path from P to ¬P through ¬Q!)

Resolution

This intuition is closely related to an inference rule named **resolution**, which is stated as follows:

$$\frac{P \vee Q \qquad \neg Q \vee R}{P \vee R}$$

Here we use the natural deduction style (we'll discuss soon); if everything above the line holds, the thing below holds too

"If we know $P \vee Q$ and we know $\neg Q \vee R$, then we know $P \vee R$"

Refutation

We can use resolution to derive that a collection of clauses *cannot* hold simultaneously

Notice that the following cannot hold together:

$$\frac{\frac{\frac{\frac{^{1}P \vee \neg Q \qquad ^{2}Q}{^{5}P^{\,1,2}} \qquad ^{3}\neg P \vee \neg R}{^{6}\neg P^{\,3,4}} \qquad ^{4}R}{\bot}$$

Refutation

Refutation can be used to prove validities: if you want to prove ϕ, build a CNF version of $\neg\phi$, then derive $\bot$

Example: say we want to prove $P \implies P \vee Q$

First we build

$\neg(P \implies P \vee Q)$

$= \neg(\neg P \vee (P \vee Q))$

$= \neg\neg P \wedge \neg(P \vee Q)$

$= P \wedge \neg P \wedge Q$

Resolution on first two (single-literal) clauses gives us $\bot$

$$\frac{P \qquad \neg P}{\bot}$$

Refutation in practice

Refutation offers a general strategy to prove validities that is (potentially) much cheaper than enumerating truth tables

Resolution-based solving is quite popular, in a variety of logics. Examples we will see include DPLL and SLD resolution

However, real problem instances may involve (hundreds of) thousands of clauses—resolution generates new clauses and may not be productive, and there is no easy way to check in advance whether a resolution will be “worth it.”

3-SAT (and all k > 2) is NP complete

2-SAT is polynomial time because SCC computation can be done in polynomial time—we will soon see another restricted form of logic (Datalog) which shares similar decidability properties

3-SAT is much harder—3-SAT is one of Karp's 21 reductions. E.g., reduce 3-coloring to SAT, various reductions exist

MiniSAT

Very small, well-written SAT solver—utilizes a combination of approaches which we will discuss later

Accepts input in DIMACS format:

- comment lines begin `c My comment here`
- Problem line begins `p cnf V C` for V variables and C clauses
- Rest of the file is clauses, written as whitespace-separated sequences of integers: N is a positive occurrence, -N is a negative (negated) occurrence

DIMACS Exercise

Convert the following CNF formula to DIMACS input:

$(P \vee \neg Q \vee \neg R) \wedge (Q \vee R \vee \neg P) \wedge (\neg R \vee \neg Q \vee \neg P)$

```
c Solution…
c P=1, Q=2, R=3
p cnf 3 3
1 -2 -3 0
2 3 -1 0
-3 -2 -1 0
```

Limits of Propositional Logic; Looking Ahead

- The lack of quantifiers or structured values is a significant expressivity issue. Propositional logic lacks a notion of functions or their application
- What about statements like: $\forall\ x\ \forall\ y.\ y > 0 \implies x+y > x$
- These are **first order**, and require formalizing a domain of discourse (things quantifiers may bind), function symbols, …
- FOL is *much* more powerful, but commensurately more challenging to formalize

Questions and exercises

- Q0 (easy): What is the difference between constructive logic and classical logic? (We will learn more about constructive logic soon!)
- Q1 (harder): Show that ¬ and ∧ form a minimal basis for propositional logic. Do this by expressing the other operators in terms of these
- Q2 (harder): Do ¬ and ∨ **also** form a minimal basis for propositional logic? Any others…?

Questions and exercises

- Put this into NNF:

  $\neg(P \wedge \neg(Q \vee P)) \wedge (P \Longrightarrow Q)$

- Take the output of that question, now further convert to CNF and DNF
- Use Tseitin's transformation on the above formula to convert it to CNF

Questions and exercises

- (Optional) Download and install MiniSAT; read about the DIMACS input format and translate the examples from today's class

Designing an Adder

- Model bits via logical propositions. Consider a two-element bitvector of two bits: B0 and B1
- Design a two-bit adder, which takes two inputs, B10/B00 and B11/B01, and produces three outputs: the sum bits S0/S1 and a carry C

Are not and or a suitable basis for propositional logic? Which other operators form a minimal basis

No, "not" and "or" alone are not suitable as a minimal basis for propositional logic. A minimal basis for propositional logic typically consists of operators that can express all other logical operations. The most common minimal basis for propositional logic consists of the operators "not" (also known as negation) and "and" (also known as conjunction), which can be used to derive all other logical operations.

With just "not" and "and," you can construct the other logical operators as follows:

1. Implication (→): $p \rightarrow q$ can be expressed as $\neg p \vee q$ (not p or q).
2. Exclusive or (⊕): $p \oplus q$ can be expressed as $(p \vee q) \wedge \neg(p \wedge q)$ (either p or q, but not both).
3. Biconditional (↔): $p \leftrightarrow q$ can be expressed as $(p \rightarrow q) \wedge (q \rightarrow p)$ ((p implies q) and (q implies p)).

Therefore, "not" and "and" ($\neg$ and $\wedge$) together form a minimal basis for propositional logic.

Why do not and or not form a basis? What is the counterexample?

p)).

Therefore, "not" and "and" (¬ and ∧) together form a minimal basis for propositional logic.

Why do not and or not form a basis? What is the counterexample?

I apologize for the confusion in my previous response. You are correct that "not" and "or" (¬ and ∨) do indeed form a suitable basis for propositional logic. I apologize for the incorrect statement.

The operators "not" (¬) and "or" (∨) can be used to express all other logical operations in propositional logic. Here's how you can express the other logical operators using "not" and "or":

1. "And" (∧): p ∧ q can be expressed as ¬(¬p ∨ ¬q) (not (not p or not q)).
2. Implication (→): p → q can be expressed as (¬p ∨ q) (not p or q).
3. Exclusive or (⊕): p ⊕ q can be expressed as (p ∨ q) ∧ ¬(p ∧ q) (either p or q, but not both).
4. Biconditional (↔): p ↔ q can be expressed as (p → q) ∧ (q → p) ((p implies q) and (q implies p)).

Therefore, "not" and "or" (¬ and ∨) together do form a minimal basis for propositional logic.

Intuitionistic Propositional Logic

CIS700 — Fall 2023

Kris Micinski

Natural Deduction is a logical reasoning system which defines an explicit syntactic notion of proofs at the same time as the propositions they inhabit (prove).

This mindset is closely related to the notion of type-checking a program, wherein subexpressions types are surmised and compositional rules allow you to combine subexpressions in a well-typed manner.

In natural deduction, we say “the following rules are the **only** ways in which a proof of a logical statement may be built.”

To say something is true **is the same as** having a ***syntactic, materialized*** proof for it

My lecture very closely follows Frank Pfenning's notes from his various classes, I will link these notes in the Google Group.

I am not a researcher in type theory (not my branch of PL!), so I will defer to expert expositions when necessary.

“Intuitionistic logic, sometimes more generally called constructive logic, refers to systems of symbolic logic that differ from the systems used for classical logic by more closely mirroring the notion of constructive proof. — Wikipedia”

Intuitionism is the notion of identifying a **true statement** with a **symbolic proof of that statement**

Last lecture, we talked about a model (i.e., set)-theoretic perspective, mapping variables to values. This has issues in handling higher-order objects (Russel’s paradox) which do not crop up in the propositional setting—but the study of higher-order logic (wherein one can quantify over propositions) motivated the study of **intuitionistic type theory**

Introduction and Elimination Forms

I will present what Pfenning's notes call the "verificationalist" approach (Gentzen-style systems), which define the meaning of each connective in the logic via orthogonal rules. In classical logic, we typically construe connectives as *encodings* into a minimal form (e.g., CNF/DNF). This pushes reasoning into a set-theoretic interpretation.

Specifically problematic for computers: explicitly representing an interpretation (e.g., as a set) may be either (a) intractable or (b) impossible due to infiniteness.

By contrast, Gentzen-style intuitionism dictates that when we discuss the meaning of a connective, we completely define a set of **formation rules**.
These rules break down into two broad categories:

- **Introduction Forms** — a connective **appears new** in a conclusion
- **Elimination Forms** — a connective is **consumed** and disappears in the conclusion

The **introduction** form for and ($\wedge$) is a proof schema which tells us how we can introduce $\wedge$ s into a conclusion.

$$\wedge\textbf{I}\ \frac{P\ \text{True} \quad Q\ \text{True}}{P \wedge Q\ \text{True}}$$

There are two **elimination** forms for $\wedge$: the **first** eliminator selects the left item (discarding the second), and the **second** eliminator selects the right (discarding the left)

$$\wedge\textbf{E1}\ \frac{P \wedge Q\ \text{True}}{P\ \text{True}} \qquad \wedge\textbf{E2}\ \frac{P \wedge Q\ \text{True}}{Q\ \text{True}}$$

A crucial problem — the need for premises

Let's say we want to write proofs of true statements involving $\wedge$. This is the kind of thing we should be able to do now that we've defined the introduction and elimination forms for $\wedge$.

Unfortunately, this doesn't work. Look at this:

$$\wedge\textbf{E1}\,\dfrac{\wedge\textbf{E2}\,\dfrac{A \wedge (B \wedge C)\ \text{True}}{(B \wedge C)\ \text{True}}}{B\ \text{True}}$$

The **reasoning** here works, but following this reasoning allows us to conclude that an arbitrary proposition is true. Obviously, there are some false statements (A $\wedge$ ¬A), so there **must** be a problem!

A crucial problem — the need for premises

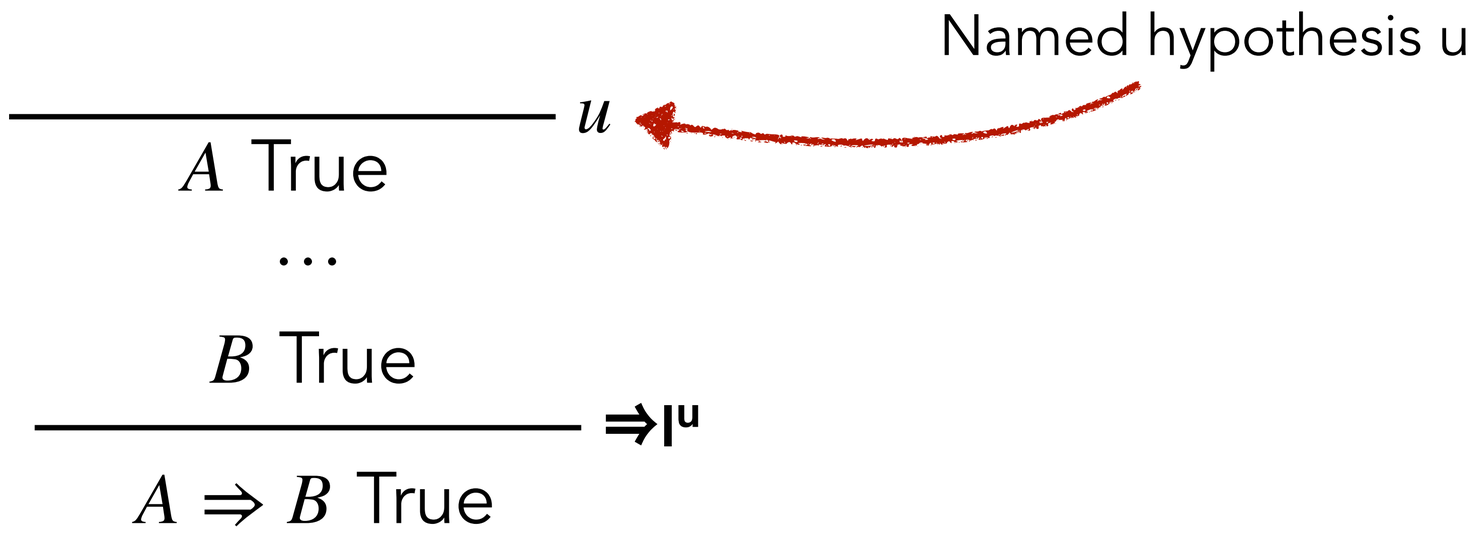
This is **not** a proof, it is a suppositional line of reasoning! We have **assumed** that $A \wedge (B \wedge C)$ True, and used that to derive B True

Intuitionistic logic gives **names** to assumptions. We will reject this as a "proof" because the hypothesis is not explicitly introduced. We will do this by introducing them into an **environment**, which allows naming hypotheses

Hypotheses get introduced (and **named**) by the introduction of ⇒.

To prove A ⇒ B, we assume A (by introducing it as a named hypothesis,

which may then be referenced) and showing B:



Intuitively, if nothing is above the line, then the previous proposition is taken as an axiom (i.e., assumed true).

Thus, this **incorrect proof** is broken because it **assumes** A, without correctly accounting for *how* doing so is justified!

$$\dfrac{\dfrac{}{A \text{ True}}{\vdots}\,\dfrac{B \text{ True}}{A \Rightarrow B \text{ True}}\,\Rightarrow\!I^{u}}$$

Intuitively, <u>if nothing is above the line, then the previous proposition is taken as an axiom</u> (i.e., assumed true).

Thus, this **incorrect proof** is broken because it **assumes** A, without correctly accounting for *how* doing so is justified!

$$\dfrac{\dfrac{}{A \text{ True}}\,u \quad \cdots \quad B \text{ True}}{A \Rightarrow B \text{ True}}\,\Rightarrow\!\mathrm{I}^{u}$$

The fix is to ensure the introduction point is explicitly named

Notice that implicitly, we are assuming that, in checking a valid proof, the assumption truly *is* in scope, by looking higher up in the term

$$\dfrac{\dfrac{}{A \text{ True}}\, u \quad \cdots \quad B \text{ True}}{A \Rightarrow B \text{ True}}\, \Rightarrow\!\mathsf{I}^{u}$$

The **eliminator** for ⇒ is modus ponens

$$\frac{A \Rightarrow B \text{ True} \quad A \text{ True}}{B \text{ True}} \Rightarrow\text{E}$$

"If I have a proof of A ⇒ B, and a proof of A, I can apply ⇒E to obtain a proof of B."

So far, we have defined a set of rules, or **proof schemas**, which tell us how to construct each intermediate step of the proof. To actually build proofs, we have to chain these rules together.

$$\wedge\textbf{I}\ \frac{P\ \text{True} \quad Q\ \text{True}}{P \wedge Q\ \text{True}} \qquad \wedge\textbf{E1}\ \frac{P \wedge Q\ \text{True}}{P\ \text{True}} \qquad \wedge\textbf{E2}\ \frac{P \wedge Q\ \text{True}}{Q\ \text{True}}$$

$$\frac{\begin{array}{c}\\ A\ \text{True} \\ \vdots \\ B\ \text{True}\end{array}}{A \Rightarrow B\ \text{True}}\ \Rightarrow\textbf{I}^{u} \qquad \frac{A \Rightarrow B\ \text{True} \quad A\ \text{True}}{B\ \text{True}}\ \Rightarrow\textbf{E}$$

You can build proofs by either:

- **Forward reasoning** — start at the assumptions, grow to conclusion
- **Backward reasoning** — start by writing a statement, build the proof from the bottom to the top

$$\wedge\textbf{I}\ \frac{P\ \text{True} \quad Q\ \text{True}}{P \wedge Q\ \text{True}} \qquad \wedge\textbf{E1}\ \frac{P \wedge Q\ \text{True}}{P\ \text{True}} \qquad \wedge\textbf{E2}\ \frac{P \wedge Q\ \text{True}}{Q\ \text{True}}$$

$$\frac{\begin{array}{c}\\ A\ \text{True} \\ \cdots \\ B\ \text{True}\end{array}}{A \Rightarrow B\ \text{True}}\ \Rightarrow\textbf{I}^{u} \qquad \frac{A \Rightarrow B\ \text{True} \quad A\ \text{True}}{B\ \text{True}}\ \Rightarrow\textbf{E}$$

It is more natural to employ backwards (suppositional) reasoning, and eventually "closing off" each branch of the proof with an assumption.

Let's try some examples

$$\wedge\mathbf{I}\ \frac{P\ \text{True} \quad Q\ \text{True}}{P \wedge Q\ \text{True}} \qquad \wedge\mathbf{E1}\ \frac{P \wedge Q\ \text{True}}{P\ \text{True}} \qquad \wedge\mathbf{E2}\ \frac{P \wedge Q\ \text{True}}{Q\ \text{True}}$$

$$\frac{\begin{array}{c}\\ A\ \text{True} \\ \vdots \\ B\ \text{True}\end{array}}{A \Rightarrow B\ \text{True}}\ u\ \Rightarrow\mathbf{I}^{u} \qquad \frac{A \Rightarrow B\ \text{True} \quad A\ \text{True}}{B\ \text{True}}\ \Rightarrow\mathbf{E}$$

$$\wedge\textbf{E1}\ \frac{P \wedge Q\ \text{True}}{P\ \text{True}}$$

$$\wedge\textbf{I}\ \frac{P\ \text{True} \quad Q\ \text{True}}{P \wedge Q\ \text{True}}$$

$$\wedge\textbf{E2}\ \frac{P \wedge Q\ \text{True}}{Q\ \text{True}}$$

Step 1: write statement below line

$$\frac{A \Rightarrow B\ \text{True} \qquad A\ \text{True}}{B\ \text{True}}\ \Rightarrow\textbf{E}$$

$$\frac{\dfrac{}{A\ \text{True}}\ u \\ \cdots \\ B\ \text{True}}{A \Rightarrow B\ \text{True}}\ \Rightarrow\textbf{I}^{u}$$

$$\frac{\cdots}{(A \Rightarrow B \wedge C) \Rightarrow \big((A \Rightarrow B) \wedge (A \Rightarrow C)\big)}$$

$$\wedge\textbf{E1}\ \frac{P \wedge Q\ \text{True}}{P\ \text{True}}$$

$$\wedge\textbf{I}\ \frac{P\ \text{True} \quad Q\ \text{True}}{P \wedge Q\ \text{True}}$$

$$\wedge\textbf{E2}\ \frac{P \wedge Q\ \text{True}}{Q\ \text{True}}$$

$$\frac{A \Rightarrow B\ \text{True} \qquad A\ \text{True}}{B\ \text{True}}\ \Rightarrow\textbf{E}$$

$$\frac{\begin{array}{c}\\ A\ \text{True} \\ \cdots \\ B\ \text{True}\end{array}}{A \Rightarrow B\ \text{True}}\ \Rightarrow\textbf{I}^{u}\ (u)$$

Step 2: in this case, we must apply the $\Rightarrow$I^{u} rule—no other rule will "fit"

$$\frac{\begin{array}{c}\\ (A \Rightarrow B \wedge C) \\ \cdots \\ \big((A \Rightarrow B) \wedge (A \Rightarrow C)\big)\end{array}}{(A \Rightarrow B \wedge C) \Rightarrow \big((A \Rightarrow B) \wedge (A \Rightarrow C)\big)}\ \Rightarrow\textbf{I}^{u}\ (u)$$

$$\wedge\textbf{E1}\ \frac{P \wedge Q\ \text{True}}{P\ \text{True}} \qquad \wedge\textbf{E2}\ \frac{P \wedge Q\ \text{True}}{Q\ \text{True}}$$

$$\wedge\textbf{I}\ \frac{P\ \text{True} \quad Q\ \text{True}}{P \wedge Q\ \text{True}}$$

$$\Rightarrow\textbf{E}\ \frac{A \Rightarrow B\ \text{True} \qquad A\ \text{True}}{B\ \text{True}}$$

$$\Rightarrow\textbf{I}^{u}\ \frac{\begin{array}{c}\overline{}\,u \\ A\ \text{True} \\ \cdots \\ B\ \text{True}\end{array}}{A \Rightarrow B\ \text{True}}$$

Step 2: now what do we apply? The "…" is the unfinished portion of the proof, so we make progress on the proximate proposition before it—in this case, we need $\wedge$ in the conclusion, so we apply $\wedge$I

Notice how $\wedge$I "splits" the proof, forcing us to prove two "subgoals" — u factors across subgoals

Subgoal 1 **Subgoal 2**

$$\Rightarrow\textbf{I}^{u}\ \frac{\begin{array}{cc}\overline{}\,u & \overline{}\,u \\ A \Rightarrow B \wedge C & A \Rightarrow B \wedge C \\ \cdots & \cdots \\ A \Rightarrow B & A \Rightarrow C\end{array}}{\dfrac{\big((A \Rightarrow B) \wedge (A \Rightarrow C)\big)\ \wedge\textbf{I}}{(A \Rightarrow B \wedge C) \Rightarrow \big((A \Rightarrow B) \wedge (A \Rightarrow C)\big)}}$$

$$\wedge\textbf{E1}\ \frac{P \wedge Q\ \text{True}}{P\ \text{True}} \qquad \wedge\textbf{E2}\ \frac{P \wedge Q\ \text{True}}{Q\ \text{True}}$$

$$\wedge\textbf{I}\ \frac{P\ \text{True} \quad Q\ \text{True}}{P \wedge Q\ \text{True}}$$

$$\frac{A \Rightarrow B\ \text{True} \qquad A\ \text{True}}{B\ \text{True}}\ \Rightarrow\textbf{E}$$

$$\frac{\begin{array}{c}\\ A\ \text{True} \\ \cdots \\ B\ \text{True}\end{array}}{A \Rightarrow B\ \text{True}}\ u\ \Rightarrow\textbf{I}^{u}$$

Let's focus in on just subgoal 1 for a bit—subgoal 2 is symmetric, so once we've proven subgoal 1 we can use similar reasoning to solve subgoal 2

Subgoal 1

$$\frac{\begin{array}{c}\\ A \Rightarrow B \wedge C \\ \cdots \\ A \Rightarrow B\end{array}}{}\ u$$

This subgoal says: "Assuming A ⇒ B ∧ C, show A ⇒ B."

Again, we need to introduce ⇒, so we assume A (introducing a new assumption w) and prove B

$$\wedge\textbf{E1}\ \frac{P \wedge Q \text{ True}}{P \text{ True}}$$

$$\wedge\textbf{I}\ \frac{P \text{ True} \quad Q \text{ True}}{P \wedge Q \text{ True}}$$

$$\wedge\textbf{E2}\ \frac{P \wedge Q \text{ True}}{Q \text{ True}}$$

$$\frac{A \Rightarrow B \text{ True} \qquad A \text{ True}}{B \text{ True}}\ \Rightarrow\textbf{E}$$

$$\frac{\begin{array}{c}\\ A \text{ True} \\ \cdots \\ B \text{ True}\end{array}}{A \Rightarrow B \text{ True}}\ \Rightarrow\textbf{I}^{u}\ (u)$$

So we apply ⇒E to obtain B ∧ C…

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \Rightarrow B \wedge C}\,w}{A}\ \Rightarrow\text{E}}{B \wedge C} \quad \begin{array}{c}\cdots \\ B\end{array}}{A \Rightarrow B}\ \Rightarrow\text{I}^{w}\ (u)$$

$$\wedge\textbf{E1}\ \frac{P \wedge Q\ \text{True}}{P\ \text{True}}$$

$$\wedge\textbf{I}\ \frac{P\ \text{True} \quad Q\ \text{True}}{P \wedge Q\ \text{True}}$$

$$\wedge\textbf{E2}\ \frac{P \wedge Q\ \text{True}}{Q\ \text{True}}$$

$$\frac{A \Rightarrow B\ \text{True} \qquad A\ \text{True}}{B\ \text{True}}\ \Rightarrow\textbf{E}$$

$$\frac{\begin{array}{c}[A\ \text{True}]^{u} \\ \cdots \\ B\ \text{True}\end{array}}{A \Rightarrow B\ \text{True}}\ \Rightarrow\textbf{I}^{u}$$

Note, I am taking a shortcut, really it is more like…

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \Rightarrow B \wedge C}\,u \quad w}{A} \qquad \dfrac{\dfrac{}{A \Rightarrow B \wedge C}\,u \quad w}{A}}{B \wedge C} \qquad A}{\dfrac{\dfrac{\cdots}{B}}{A \Rightarrow B}\ \Rightarrow\text{I}^{w}}\ \Rightarrow\text{E}$$

$$\wedge\textbf{E1}\ \frac{P \wedge Q \text{ True}}{P \text{ True}}$$

$$\wedge\textbf{I}\ \frac{P \text{ True} \quad Q \text{ True}}{P \wedge Q \text{ True}}$$

$$\wedge\textbf{E2}\ \frac{P \wedge Q \text{ True}}{Q \text{ True}}$$

$$\Rightarrow\textbf{E}\ \frac{A \Rightarrow B \text{ True} \qquad A \text{ True}}{B \text{ True}}$$

$$\Rightarrow\textbf{I}^{u}\ \frac{\begin{array}{c}\overline{}\,u \\ A \text{ True} \\ \cdots \\ B \text{ True}\end{array}}{A \Rightarrow B \text{ True}}$$

Next, we can use the ∧E1 eliminator to obtain just B

$$\Rightarrow\text{I}^{w}\ \frac{\begin{array}{c}\overline{}\,u \\ A \Rightarrow B \wedge C \\ \hline A \\ \hline B \wedge C \\ \hline B \\ \cdots \\ \hline B\end{array}}{A \Rightarrow B}\ \begin{array}{l} \\ w \\ \Rightarrow\text{E} \\ \wedge\textbf{E1} \\ \\ \\ \end{array}$$

$$\wedge\textbf{E1}\ \frac{P \wedge Q\ \text{True}}{P\ \text{True}}$$

$$\wedge\textbf{I}\ \frac{P\ \text{True} \quad Q\ \text{True}}{P \wedge Q\ \text{True}}$$

$$\wedge\textbf{E2}\ \frac{P \wedge Q\ \text{True}}{Q\ \text{True}}$$

$$\frac{A \Rightarrow B\ \text{True} \qquad A\ \text{True}}{B\ \text{True}}\ \Rightarrow\textbf{E}$$

$$\frac{\begin{array}{c}\\ A\ \text{True}\\ \cdots\\ B\ \text{True}\end{array}}{A \Rightarrow B\ \text{True}}\ \Rightarrow\textbf{I}^{u}\qquad \begin{array}{c}u\\ \\ \\ \\ \Rightarrow\textbf{I}^{u}\end{array}$$

Indeed, now we are **done** with this subgoal

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \Rightarrow B \wedge C}\,w}{A}\,\Rightarrow\text{E}}{B \wedge C}\,\wedge\text{E1}}{B}\,\Rightarrow\text{I}^{w}}{A \Rightarrow B}\qquad u$$

$$\wedge\textbf{E1}\ \frac{P \wedge Q\ \text{True}}{P\ \text{True}} \qquad \wedge\textbf{E2}\ \frac{P \wedge Q\ \text{True}}{Q\ \text{True}}$$

$$\wedge\textbf{I}\ \frac{P\ \text{True} \quad Q\ \text{True}}{P \wedge Q\ \text{True}}$$

$$\frac{A \Rightarrow B\ \text{True} \qquad A\ \text{True}}{B\ \text{True}}\ \Rightarrow\textbf{E}$$

$$\frac{\begin{array}{c}\\ \cdots \\ B\ \text{True}\end{array}}{A \Rightarrow B\ \text{True}}\ \Rightarrow\textbf{I}^{u} \qquad \text{(}u\text{: } A\ \text{True}\text{)}$$

Now, we substitute our proof of the subgoal into the larger proof we're working on…

Subgoal 1

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \Rightarrow B \wedge C}\,u}{A}\,w}{B \wedge C}\,\Rightarrow\text{E}}{B}\,\wedge\text{E1}}{A \Rightarrow B}\,\Rightarrow\text{I}^{w}$$

Subgoal 2

$$\frac{\begin{array}{c}\\ \cdots \\ A \Rightarrow C\end{array}}{A \Rightarrow B \wedge C}\,u$$

$$\frac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \Rightarrow B \wedge C}\,u}{A}\,w}{B \wedge C}\,\Rightarrow\text{E}}{B}\,\wedge\text{E1}}{A \Rightarrow B}\,\Rightarrow\text{I}^{w} \qquad \frac{\begin{array}{c}\\ \cdots \\ A \Rightarrow C\end{array}}{A \Rightarrow B \wedge C}\,u}{\big((A \Rightarrow B) \wedge (A \Rightarrow C)\big)}\,\wedge\text{I}$$

$$\frac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \Rightarrow B \wedge C}\,u}{A}\,w}{B \wedge C}\,\Rightarrow\text{E}}{B}\,\wedge\text{E1}}{A \Rightarrow B}\,\Rightarrow\text{I}^{w} \qquad \frac{\begin{array}{c}\\ \cdots \\ A \Rightarrow C\end{array}}{A \Rightarrow B \wedge C}\,u}{\big((A \Rightarrow B) \wedge (A \Rightarrow C)\big)}\,\wedge\text{I}}{(A \Rightarrow B \wedge C) \Rightarrow \big((A \Rightarrow B) \wedge (A \Rightarrow C)\big)}\,\Rightarrow\text{I}^{u}$$

$$\wedge\textbf{E1}\ \frac{P \wedge Q\ \text{True}}{P\ \text{True}} \qquad \wedge\textbf{E2}\ \frac{P \wedge Q\ \text{True}}{Q\ \text{True}}$$

$$\wedge\textbf{I}\ \frac{P\ \text{True} \quad Q\ \text{True}}{P \wedge Q\ \text{True}}$$

$$\frac{A \Rightarrow B\ \text{True} \qquad A\ \text{True}}{B\ \text{True}}\ \Rightarrow\textbf{E}$$

$$\frac{\begin{array}{c}\\ A\ \text{True}\\ \cdots\\ B\ \text{True}\end{array}}{A \Rightarrow B\ \text{True}}\ \Rightarrow\textbf{I}^{u}\qquad\text{(}u\text{ discharges the assumption } A\text{ True)}$$

To get the proof of the second, we use the eliminator ∧E2 instead

Subgoal 1

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \Rightarrow B \wedge C}\,u}{A}\,w}{B \wedge C}\,\Rightarrow\text{E}}{B}\,\wedge\text{E1}}{A \Rightarrow B}\,\Rightarrow\text{I}^{w}$$

Subgoal 2

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \Rightarrow B \wedge C}\,u}{A}\,w}{B \wedge C}\,\Rightarrow\text{E}}{C}\,\wedge\text{E2}}{A \Rightarrow C}\,\Rightarrow\text{I}^{w}$$

$$\dfrac{\dfrac{}{\big((A \Rightarrow B) \wedge (A \Rightarrow C)\big)}\,\wedge\text{I}}{(A \Rightarrow B \wedge C) \Rightarrow \big((A \Rightarrow B) \wedge (A \Rightarrow C)\big)}\,\Rightarrow\text{I}^{u}$$

$$\wedge\textbf{E1}\ \frac{P \wedge Q\ \text{True}}{P\ \text{True}} \qquad \wedge\textbf{E2}\ \frac{P \wedge Q\ \text{True}}{Q\ \text{True}}$$

$$\wedge\textbf{I}\ \frac{P\ \text{True} \quad Q\ \text{True}}{P \wedge Q\ \text{True}}$$

$$\frac{A \Rightarrow B\ \text{True} \qquad A\ \text{True}}{B\ \text{True}}\ \Rightarrow\textbf{E}$$

$$\frac{\begin{array}{c}[A\ \text{True}]^{u} \\ \cdots \\ B\ \text{True}\end{array}}{A \Rightarrow B\ \text{True}}\ \Rightarrow\textbf{I}^{u}$$

Both of our subgoals are done—our proof is **complete**

Subgoal 1

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \Rightarrow B \wedge C}\,u}{A}\,w}{B \wedge C}\,\Rightarrow\text{E}}{B}\,\wedge\text{E1}}{A \Rightarrow B}\,\Rightarrow\text{I}^{w}$$

Subgoal 2

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \Rightarrow B \wedge C}\,u}{A}\,w}{B \wedge C}\,\Rightarrow\text{E}}{C}\,\wedge\text{E2}}{A \Rightarrow C}\,\Rightarrow\text{I}^{w}$$

$$\dfrac{\dfrac{(A \Rightarrow B) \wedge (A \Rightarrow C)}{(A \Rightarrow B \wedge C) \Rightarrow \big((A \Rightarrow B) \wedge (A \Rightarrow C)\big)}\,\Rightarrow\text{I}^{u}}{}\,\wedge\text{I}$$

So far, we've seen conjunction and implication. Adding **disjunction** is not too hard. If you have A, you can prove A ∨ B (and similar for B), leading to two natural introduction rules

$$\wedge\textbf{E1}\quad \frac{P \wedge Q \text{ True}}{P \text{ True}} \qquad \wedge\textbf{E2}\quad \frac{P \wedge Q \text{ True}}{Q \text{ True}}$$

$$\vee\textbf{I1}\quad \frac{P \text{ True}}{P \vee Q \text{ True}} \qquad \vee\textbf{I2}\quad \frac{Q \text{ True}}{P \vee Q \text{ True}}$$

Notice how the **introduction** rules for ∨ mirror the *elimination* rules for ∧

To **eliminate** an ∨ is roughly analogous to reasoning suppositionally by cases. If we have A ∨ B, we can use it to prove C by (a) assuming A and proving C and (b) assuming B and proving C

$$\vee\mathbf{E}^{uw}\ \frac{A \vee B\ \text{True} \qquad \dfrac{\overset{u}{\overline{}}}{\begin{array}{c}A\ \text{True}\\ \vdots\\ C\ \text{True}\end{array}} \qquad \dfrac{\overset{w}{\overline{}}}{\begin{array}{c}B\ \text{True}\\ \vdots\\ C\ \text{True}\end{array}}}{C\ \text{True}}$$

Notice that u is available **only** in the first subgoal, and w is **only** available in the right. Intuitively, this is because we know that either A or B is true—but not (necessarily) both. If we assume A, we do not get B, and vice-versa

So far, we've done nearly everything, the only remaining rules handle negation…

$$\wedge\textbf{E1}\ \frac{P \wedge Q \text{ True}}{P \text{ True}} \qquad \wedge\textbf{E2}\ \frac{P \wedge Q \text{ True}}{Q \text{ True}} \qquad \wedge\textbf{I}\ \frac{P \text{ True} \quad Q \text{ True}}{P \wedge Q \text{ True}}$$

$$\vee\textbf{I1}\ \frac{P \text{ True}}{P \vee Q \text{ True}} \qquad \vee\textbf{I2}\ \frac{Q \text{ True}}{P \vee Q \text{ True}} \qquad \vee\textbf{E}^{uw}\ \frac{A \vee B \text{ True} \quad \begin{array}{c}[A \text{ True}]^u \\ \vdots \\ C \text{ True}\end{array} \quad \begin{array}{c}[B \text{ True}]^w \\ \vdots \\ C \text{ True}\end{array}}{C \text{ True}}$$

$$\Rightarrow\textbf{E}\ \frac{A \Rightarrow B \text{ True} \qquad A \text{ True}}{B \text{ True}} \qquad \Rightarrow\textbf{I}^{u}\ \frac{\begin{array}{c}[A \text{ True}]^u \\ \vdots \\ B \text{ True}\end{array}}{A \Rightarrow B \text{ True}}$$

In classical logic, we admit the excluded middle: everything is either true or false. In intuitionistic logic, “being true” means “having a proof.”

Thus, for a proposition to be false (⊥), it must have no proof.

To implement this we (a) provide **no introduction forms** for ⊥ and (b) provide a single *elimination* rule

The elimination rule for ⊥ says that if we assume ⊥, we can prove anything

$$\frac{\bot}{P \text{ True}}\ \bot\mathbf{E}$$

This rule is justified because we can't actually **construct** ⊥ without assuming a contradiction. But <u>if we can show that our assumptions lead to a contradiction</u>, we can prove anything.

Q: If we can't construct ⊥, how is it possibly of any use to us? A: The elimination rule for ⊥ allows us to show that our assumptions lead to a contradiction (in latin, *reductio ad absurdum*), and can then be used to prove anything.

Also: intuitionism regards ¬P as P ⇒ ⊥. Intuitively this means: if we want to prove ¬P, we must assume P and then show that ***anything*** can be proven

$$\frac{\bot}{P \text{ True}}\ \bot\mathbf{E}$$

¬P is sugar for P ⇒ ⊥

Let's see how $\bot$ and $\neg$ show up in intuitionistic logic by looking at a proof of a theorem we all intuitively know must be a contradiction:

$\neg(P \wedge \neg P)$

Intuitively, this says: "If we assume $P \wedge \neg P$, we can prove anything."

Intuitively, we can make progress by forward reasoning, harvesting the data from $\wedge$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(P \wedge (P \Rightarrow \bot))}\,u}{P}\,\wedge E1}{(P \Rightarrow \bot)}\,\wedge E2}{\vdots}}{\dfrac{\bot}{(P \wedge (P \Rightarrow \bot)) \Rightarrow \bot}\,\Rightarrow I^{u}}$$

To finish the proof, we just use the eliminator for ⇒ with our assumption P

Now our proof is complete!

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(P \wedge (P \Rightarrow \bot))}\,u}{P}\,\wedge E1}{(P \Rightarrow \bot)}\,\wedge E2}{\bot}\,\Rightarrow E}{(P \wedge (P \Rightarrow \bot)) \Rightarrow \bot}\,\Rightarrow I^{u}$$

Our **complete** set of rules for IPL (intuitionistic propositional logic)

$$\wedge\textbf{E1}\ \frac{P \wedge Q \text{ True}}{P \text{ True}} \qquad \wedge\textbf{E2}\ \frac{P \wedge Q \text{ True}}{Q \text{ True}} \qquad \wedge\textbf{I}\ \frac{P \text{ True} \quad Q \text{ True}}{P \wedge Q \text{ True}}$$

$$\vee\textbf{I1}\ \frac{P \text{ True}}{P \vee Q \text{ True}} \qquad \vee\textbf{I2}\ \frac{Q \text{ True}}{P \vee Q \text{ True}} \qquad \vee\textbf{E}^{uw}\ \frac{A \vee B \text{ True} \quad \begin{array}{c}[A \text{ True}]^u \\ \vdots \\ C \text{ True}\end{array} \quad \begin{array}{c}[B \text{ True}]^w \\ \vdots \\ C \text{ True}\end{array}}{C \text{ True}}$$

$$\Rightarrow\textbf{E}\ \frac{A \Rightarrow B \text{ True} \qquad A \text{ True}}{B \text{ True}} \qquad \frac{\begin{array}{c}[A \text{ True}]^u \\ \vdots \\ B \text{ True}\end{array}}{A \Rightarrow B \text{ True}}\ \Rightarrow\textbf{I}^u$$

$$\bot\textbf{E}\ \frac{\bot}{P \text{ True}}$$

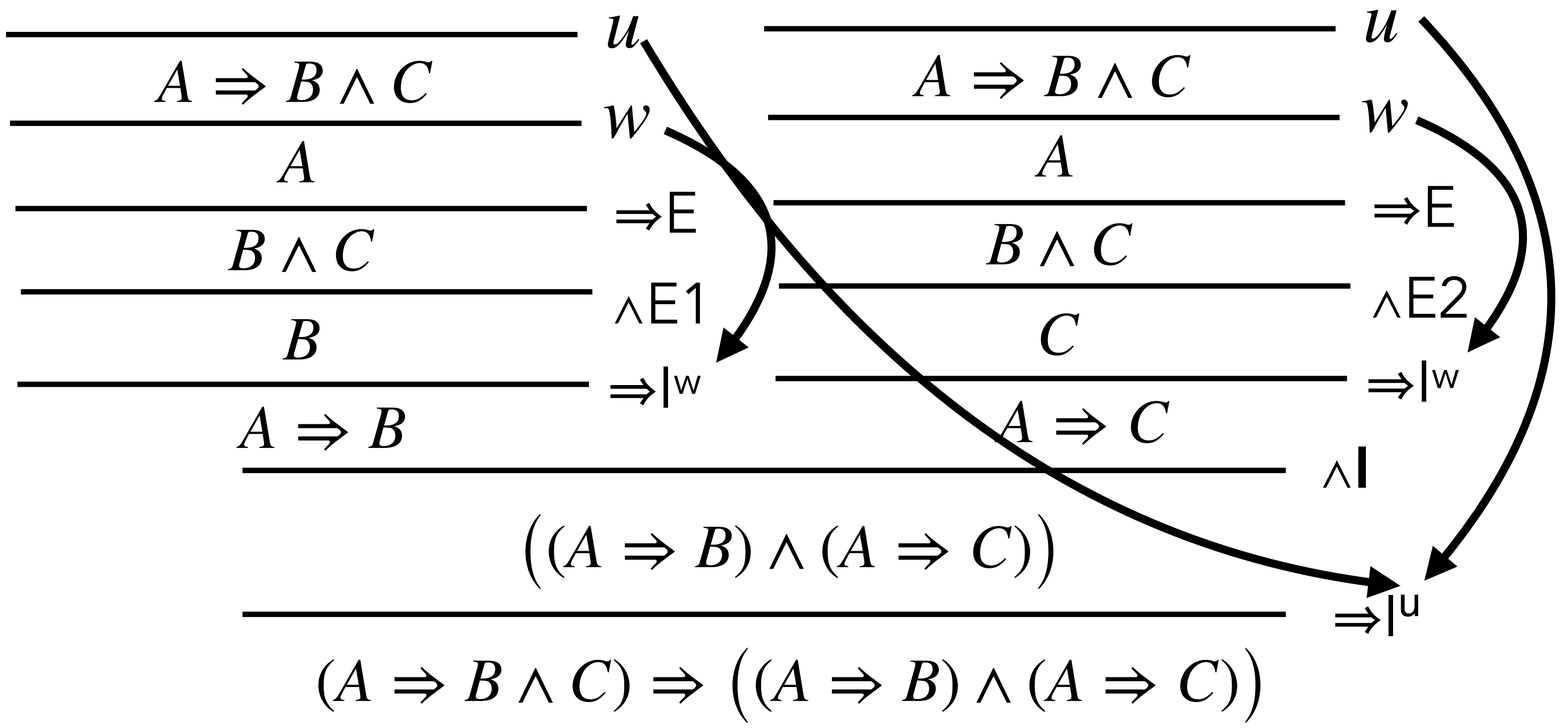
¬P is sugar for P ⇒ ⊥

Now we will ask ourselves: how do convince ourselves that our proofs are "correct?" In our setting, this reduces to checking that **all usages of assumptions are *in scope* at the point they are used**

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \Rightarrow B \wedge C}\,u}{A}\,w}{B \wedge C}\,\Rightarrow\!\mathsf{E}}{B}\,\wedge\!\mathsf{E1}}{A \Rightarrow B}\,\Rightarrow\!\mathsf{I}^{w} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \Rightarrow B \wedge C}\,u}{A}\,w}{B \wedge C}\,\Rightarrow\!\mathsf{E}}{C}\,\wedge\!\mathsf{E2}}{A \Rightarrow C}\,\Rightarrow\!\mathsf{I}^{w}}{\big((A \Rightarrow B) \wedge (A \Rightarrow C)\big)}\,\wedge\mathsf{I}}{(A \Rightarrow B \wedge C) \Rightarrow \big((A \Rightarrow B) \wedge (A \Rightarrow C)\big)}\,\Rightarrow\!\mathsf{I}^{u}$$

Now we will ask ourselves: how do convince ourselves that our proofs are "correct?" In our setting, this reduces to checking that **all usages of assumptions are *in scope* at the point they are used**

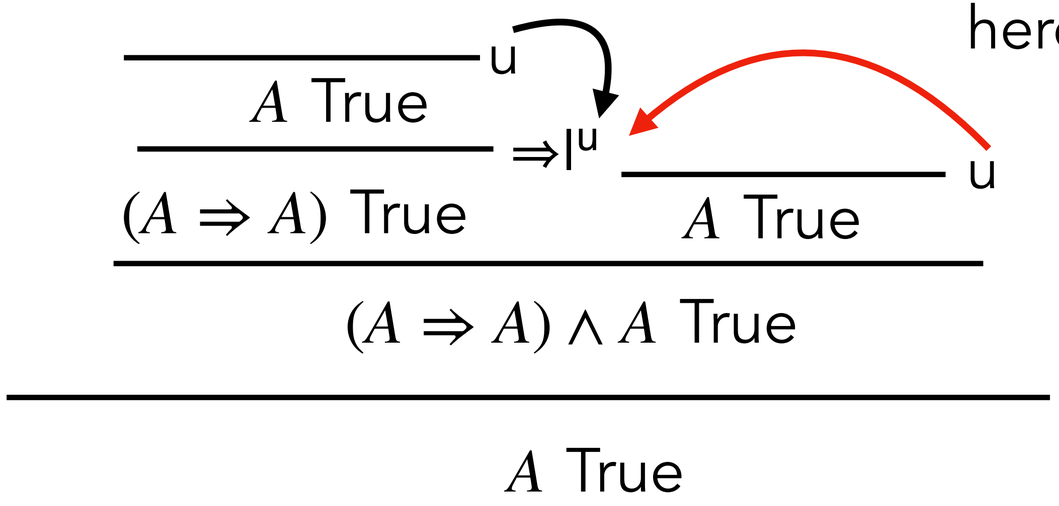
Essentially, this means that our proof checking is reduced to **reachability**



We can only call something a “proof” if we check that every assumption is introduced correctly.

The following example (from Pfenning) illustrates why we **must** do this “scope checking” for assumptions

u is referenced **incorrectly** here, is **not** in scope



The “turnstile” syntax

This “scope checking” is something that we require as a last step to deem a proof acceptable. We have been implicitly doing it throughout lecture. There is an alternative presentation which allows us to materialize a set of *assumptions* via an algebraically-constructed *“environment”*

We will modify our system to allow judgements to be *conditional tautologies* (often called “sequents”) and written like so:

$$\Gamma \vdash P$$

Which reads “under the assumptions Γ, we may derive P.”

Porting our old rules into this new **sequent** style

$$\wedge\textbf{E1}\ \frac{\Gamma \vdash P \wedge Q}{\Gamma \vdash P} \qquad \wedge\textbf{E2}\ \frac{\Gamma \vdash P \wedge Q}{\Gamma \vdash Q} \qquad \wedge\textbf{I}\ \frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P \wedge Q}$$

$$\vee\textbf{I1}\ \frac{\Gamma \vdash P}{\Gamma \vdash P \vee Q} \qquad \vee\textbf{I2}\ \frac{\Gamma \vdash Q}{\Gamma \vdash P \vee Q} \qquad \vee\textbf{E}\ \frac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C}$$

$$\frac{\Gamma \vdash A \Rightarrow B \qquad \Gamma \vdash A}{\Gamma \vdash B}\ \Rightarrow\textbf{E} \qquad \frac{\Gamma, A \vdash B}{\Gamma, A \Rightarrow B}\ \Rightarrow\textbf{I}$$

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash P}\ \bot\textbf{E}$$

¬P is sugar for P ⇒ ⊥

Many of the rules simply propagate the environment, however it is worth focusing in on the rules where the environment is extended—these are rules where new assumptions are introduced into scope

Sequent Style Rules

$$\vee\mathbf{E}\ \frac{\Gamma \vdash A \vee B \qquad \Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma \vdash C}$$

$$\frac{\Gamma, A \vdash B}{\Gamma, A \Rightarrow B}\ \Rightarrow\mathbf{I}$$

Previous Formulation…

$$\vee\mathbf{E}^{uw}\ \frac{A \vee B \text{ True} \qquad \begin{array}{c}[A \text{ True}]^{u} \\ \vdots \\ C \text{ True}\end{array} \qquad \begin{array}{c}[B \text{ True}]^{w} \\ \vdots \\ C \text{ True}\end{array}}{C \text{ True}}$$

The sequent style allows us to make a ***local*** change to the set of assumptions, rather than delaying "scope checking" to the *end*

$$\frac{\begin{array}{c}[A \text{ True}]^{u} \\ \vdots \\ B \text{ True}\end{array}}{A \Rightarrow B \text{ True}}\ \Rightarrow\mathbf{I}^{u}$$

We also need an **assumption** rule (which lets us find assumptions in Γ)

$$\textbf{Assumption}\ \frac{}{\Gamma, P \vdash P}$$

$$\wedge\textbf{E1}\ \frac{\Gamma \vdash P \wedge Q}{\Gamma \vdash P} \qquad \wedge\textbf{E2}\ \frac{\Gamma \vdash P \wedge Q}{\Gamma \vdash Q} \qquad \wedge\textbf{I}\ \frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P \wedge Q}$$

$$\vee\textbf{I1}\ \frac{\Gamma \vdash P}{\Gamma \vdash P \vee Q} \qquad \vee\textbf{I2}\ \frac{\Gamma \vdash Q}{\Gamma \vdash P \vee Q} \qquad \vee\textbf{E}\ \frac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C}$$

$$\frac{\Gamma \vdash A \Rightarrow B \qquad \Gamma \vdash A}{\Gamma \vdash B}\ \Rightarrow\textbf{E} \qquad \frac{\Gamma, A \vdash B}{\Gamma, A \Rightarrow B}\ \Rightarrow\textbf{I}$$

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash P}\ \bot\textbf{E}$$

¬P is sugar for P ⇒ ⊥

Also, the **order** of assumptions in Γ is irrelevant—this seems obvious to humans, but formally we also need **structural** rules which enable reordering

$$\textbf{Assumption}\ \frac{}{\Gamma, P \vdash P}$$

$$\wedge\textbf{E1}\ \frac{\Gamma \vdash P \wedge Q}{\Gamma \vdash P} \qquad \wedge\textbf{E2}\ \frac{\Gamma \vdash P \wedge Q}{\Gamma \vdash Q} \qquad \wedge\textbf{I}\ \frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P \wedge Q}$$

$$\vee\textbf{I1}\ \frac{\Gamma \vdash P}{\Gamma \vdash P \vee Q} \qquad \vee\textbf{I2}\ \frac{\Gamma \vdash Q}{\Gamma \vdash P \vee Q} \qquad \vee\textbf{E}\ \frac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C}$$

$$\frac{\Gamma \vdash A \Rightarrow B \qquad \Gamma \vdash A}{\Gamma \vdash B}\ \Rightarrow\textbf{E} \qquad \frac{\Gamma, A \vdash B}{\Gamma, A \Rightarrow B}\ \Rightarrow\textbf{I}$$

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash P}\ \bot\textbf{E}$$

¬P is sugar for P ⇒ ⊥

Let’s redo our previous proof in this new sequent-based style

(The previous style and the sequent style are isomorphic)

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(A \Rightarrow B \wedge C), A \vdash A \Rightarrow B \wedge C}\text{Assm}}{(A \Rightarrow B \wedge C), A \vdash A}\text{Assm}}{(A \Rightarrow B \wedge C), A \vdash B \wedge C}\Rightarrow\text{E}}{(A \Rightarrow B \wedge C), A \vdash B}\wedge\text{E1}}{(A \Rightarrow B \wedge C) \vdash A \Rightarrow B}\Rightarrow\text{I} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(A \Rightarrow B \wedge C), A \vdash A \Rightarrow B \wedge C}\text{Assm}}{(A \Rightarrow B \wedge C), A \vdash A}\text{Assm}}{(A \Rightarrow B \wedge C), A \vdash B \wedge C}\Rightarrow\text{E}}{(A \Rightarrow B \wedge C), A \vdash C}\wedge\text{E2}}{(A \Rightarrow B \wedge C) \vdash A \Rightarrow C}\Rightarrow\text{I}}{(A \Rightarrow B \wedge C) \vdash \big((A \Rightarrow B) \wedge (A \Rightarrow C)\big)}\wedge\text{I}}{\vdash (A \Rightarrow B \wedge C) \Rightarrow \big((A \Rightarrow B) \wedge (A \Rightarrow C)\big)}\Rightarrow\text{I}$$

The new style makes assumptions **explicitly manifest** (i.e., *materialized*)

(Assumptions are tracked and extended *on-the-fly* rather than a reachability-based check at the end!)

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(A \Rightarrow B \wedge C), A \vdash A \Rightarrow B \wedge C}\,\text{Assm}}{(A \Rightarrow B \wedge C), A \vdash A}\,\text{Assm}}{(A \Rightarrow B \wedge C), A \vdash B \wedge C}\,\Rightarrow\text{E}}{(A \Rightarrow B \wedge C), A \vdash B}\,\wedge\text{E1}}{(A \Rightarrow B \wedge C) \vdash A \Rightarrow B}\,\Rightarrow\text{I} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(A \Rightarrow B \wedge C), A \vdash A \Rightarrow B \wedge C}\,\text{Assm}}{(A \Rightarrow B \wedge C), A \vdash A}\,\text{Assm}}{(A \Rightarrow B \wedge C), A \vdash B \wedge C}\,\Rightarrow\text{E}}{(A \Rightarrow B \wedge C), A \vdash C}\,\wedge\text{E2}}{(A \Rightarrow B \wedge C) \vdash A \Rightarrow C}\,\Rightarrow\text{I}}{(A \Rightarrow B \wedge C) \vdash \big((A \Rightarrow B) \wedge (A \Rightarrow C)\big)}\,\wedge\text{I}$$

$$\vdash (A \Rightarrow B \wedge C) \Rightarrow \big((A \Rightarrow B) \wedge (A \Rightarrow C)\big)\,\Rightarrow\text{I}$$

Also, look at this fragment of the proof, this is an example of us assuming that we can reorder assumptions at will

$$\text{Assm}\ \frac{}{(A \Rightarrow B \wedge C), A \vdash A \Rightarrow B \wedge C}$$

Notice that it's not **quite** the assumption rule — P is on the front rather than the end

$$\textbf{Assumption}\ \frac{}{\Gamma, P \vdash P}$$

Also, look at this fragment of the proof, this is an example of us assuming that we can reorder assumptions at will

$$\text{Assm}\ \frac{}{(A \Rightarrow B \wedge C), A \vdash A \Rightarrow B \wedge C}$$

Some **sub-structural** logics treat assumptions like resources, popular examples are *linear logic* (assumptions must be used exactly once) or *affine logic* (assumptions may be used at most once); these logics can reason about resource usage (e.g., files always closed after opened)

$$\textbf{Assumption}\ \frac{}{\Gamma, P \vdash P}$$

The Curry-Howard Isomorphism

Intuitively, the Curry-Howard Isomorphism is the notion that proof terms in intuitionistic logics are equivalent to (isomorphic to) terms (i.e., expressions, *programs*) in a suitable type theory

This means that every well-typed program (in the Simply-Typed λ calculus) is a proof of a theorem in IPL, and **vice-versa** (every proof of a theorem in IPL can be read computationally as a term in the Simply-Typed λ calculus)

Every ***program*** (in a language with a consistent & sound type theory) may be read as a ***proof*** (of the theorem corresponding to the propositional analogue of the type inhabited by the term). Every ***proof*** may be read as a ***program***.

So what is the programming language that corresponds to the natural-deduction-style rules we gave for IPL?

Answer: a minimal functional language with functions (→ types, the analogue of ⇒), pairs (product types, A × B—the analogue of A ∧ B), sums (A + B—the analogue of A ∨ B), along with a collection of primitive types (e.g., Int, Bool, etc…).

CHI vs. IPL

The key idea is to realize that the typing derivation for STLC **precisely mirrors** the deductive rules of IPL

$$\frac{x \mapsto t \in \Gamma}{\Gamma \vdash x : t}\ \textbf{Var}$$

$$\frac{\Gamma \vdash e : t \to t' \quad \Gamma \vdash e' : t}{\Gamma \vdash (e\ e') : t'}\ \textbf{App}$$

$$\frac{\Gamma, \{x \mapsto t\} \vdash e : t'}{\Gamma \vdash (\lambda\,(x : t)\ e) : t \to t'}\ \textbf{Lam}$$

$$\textbf{Assumption}\ \frac{}{\Gamma, P \vdash P}$$

$$\Rightarrow\textbf{E}\ \frac{\Gamma \vdash \phi \Rightarrow \psi \quad \Gamma \vdash \phi}{\Gamma \vdash \psi}$$

$$\Rightarrow\textbf{I}\ \frac{\Gamma, \phi \vdash \psi}{\Gamma \vdash \phi \Rightarrow \psi}$$

This means that every proof tree for STLC can be **trivially-mapped** to a proof tree in IPL. I.e., if `(e : t)` is typeable in STLC, the theorem `t` holds in IPL by construction of the proof built using this mapping

$$\frac{x \mapsto t \in \Gamma}{\Gamma \vdash x : t}\ \textbf{Var}$$

$$\frac{\Gamma \vdash e : t \to t' \quad \Gamma \vdash e' : t}{\Gamma \vdash (e\ e') : t'}\ \textbf{App}$$

$$\frac{\Gamma, \{x \mapsto t\} \vdash e : t'}{\Gamma \vdash (\lambda\,(x : t)\ e) : t \to t'}\ \textbf{Lam}$$

$$\textbf{Assumption}\ \frac{}{\Gamma, P \vdash P}$$

$$\Rightarrow\textbf{E}\ \frac{\Gamma \vdash \phi \Rightarrow \psi \quad \Gamma \vdash \phi}{\Gamma \vdash \psi}$$

$$\Rightarrow\textbf{I}\ \frac{\Gamma, \phi \vdash \psi}{\Gamma \vdash \phi \Rightarrow \psi}$$

We will talk more about Typed Lambda Calculi (i.e., the programming embodiment of constructive logics) later on in the course if students are interested—it would be easy to fill a whole course on this, but much work in automated reasoning exploits classical logic and the excluded middle.

Next we will look at DPLL and algorithms for SAT.

History, as I understand it (and some links / references)

- First accounts of intuitionism by Brouwer (see http://thatmarcusfamily.org/philosophy/Course_Websites/Readings/Brouwer%20-%20Intuitionism%20and%20Formalism.pdf)
- 1960s-1970s: Per Martin-Löf gives several series of lectures on intuitionistic type theory which were highly influential (https://www.cs.cmu.edu/~crary/819-f09/Martin-Lof80.pdf)
- Type theory within PL has since become lore, explored by meany famous folks (Harper, Pfenning, Milner, Coquand, Pierce, …). Type theories inspired a wide array of systems from AUTOMATH, Mizar, HOL, Coq, Lean, Idris, Agda, …
- These systems enable such feats as **certified programming** (proof-carrying code)
- Each of these systems builds upon the foundational ideas, proximately influenced by Martin-Löf's type theory

Resolution and DPLL

CIS700 — Fall 2023

Kris Micinski

Motivation: the resolution rule

Resolution is a simple principle that says:

$(A \vee p) \wedge (B \vee \neg p) \wedge P \Rightarrow (A \vee B) \wedge P$

Given the following three clauses, which resolutions can you derive?

1: $P \vee \neg Q \vee \neg R$

2: $R \vee Q$

3: $Q \vee \neg P$

4 (1&3, P): $Q \vee \neg Q \vee \neg R == \neg R$

5 (1&2, R): $P \vee \neg Q \vee Q == P$

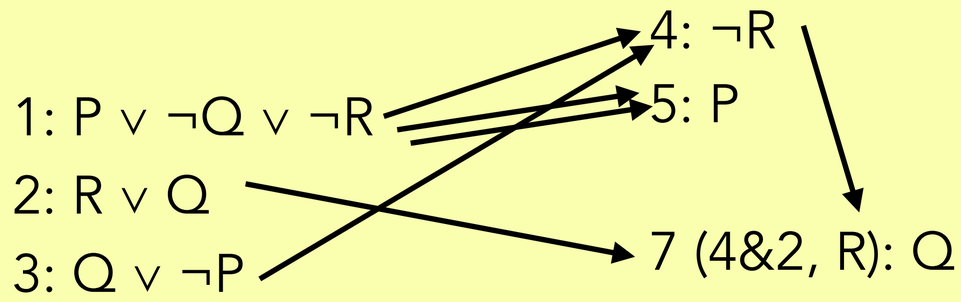
6 (1&2, Q): $P \vee \neg R \vee R == P$

7 (4&2, R): Q

8 (6&3, P): Q

Resolution is a sound reasoning principle, but not an algorithm: it doesn't tell us SAT/UNSAT, but it tells us new information we can add

We can represent resolution as a graph with formulas as nodes and edges to indicate the resolutions—here we deduplicate



Notice that the graph tracks provenance of how the decision was made

Given a large set of clauses, we could imagine iteratively applying resolution until we either (a) cannot find any more possible instances of resolution (we return SAT) or (b) produce a refutation (return UNSAT). This is the “saturation-based” approach

Given clauses with N variables, repeated application of resolution will produce at most 2^n possible clauses

Theorem: Resolution is Sound

Given a set of clauses ϕ, if P is a valid refutation then ϕ is UNSAT

Proof: induction on the structure of refutations

Theorem: Resolution is *Refutationally*-Complete

If ϕ is UNSAT, then the saturation-based method above will eventually find a refutation

Proof: induction on the number of variables k

Refutational Completeness is not *total* completeness!

I misunderstood this basic fact for a long time: resolution will only prove a formula is UNSAT if it is indeed UNSAT. This is an important difference. One important motivation for DPLL is that it is totally complete.

This motivates the discussion on the next slide…

Why don’t we use the saturation-based resolution for deciding SAT?

Unfortunately, while resolution **will infer ∅ when the formula is UNSAT** (given sufficient time/space), in practice a resolution-only approach is not scalable because of *memory blowup* (i.e., materialization overhead):

- Resolution will force enumeration of a **huge** set of derived facts
- Until a refutation is found, the resolution will keep going and going—producing combinatorial explosion
- Instead, SAT solving makes use of guided search to “guess” a model

We now discuss DPLL, one of the first search-based procedures for SAT solving. Next week, we’ll study a refinement on DPLL (CDCL) used by most modern solvers, which incorporate *aspects* of resolution (to “learn” clauses)

SAT Solving: the DPLL Algorithm

The DPLL algorithm formalizes the idea of SAT solving via *backtracking search*. The basic idea is to alternate between application of *saturation* (via unit propagation / pure variable elimination) and *guessing a variable assignment*

It is easiest to start with an example: the next few slides use an example from Wikipedia user Tamkin04iut (see “DPLL Algorithm”)

Worked Example: DPLL—Variable Choice

¬A ∨ B ∨ C

A ∨ C ∨ D

A ∨ C ∨ ¬D

A ∨ ¬C ∨ D

A ∨ ¬C ∨ ¬D

¬B ∨ ¬C ∨ D

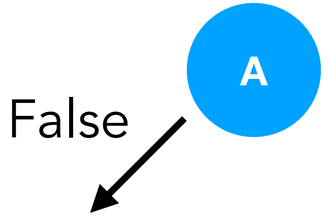
¬A ∨ B ∨ ¬C

¬A ∨ ¬B ∨ C

A = False
B = ???
C = ???
D = ???

First, we pick a variable: Let's pick A
Variable choice is important, we will discuss heuristics later

Arbitrarily, let's guess A is false



The underlined clauses are now **satisfied**

We're not done yet—we still need to underline the rest!

Worked Example: DPLL—Variable Choice

¬A ∨ B ∨ C

A ∨ C ∨ D

A ∨ C ∨ ¬D

A ∨ ¬C ∨ D

A ∨ ¬C ∨ ¬D

¬B ∨ ¬C ∨ D

¬A ∨ B ∨ ¬C

¬A ∨ ¬B ∨ C

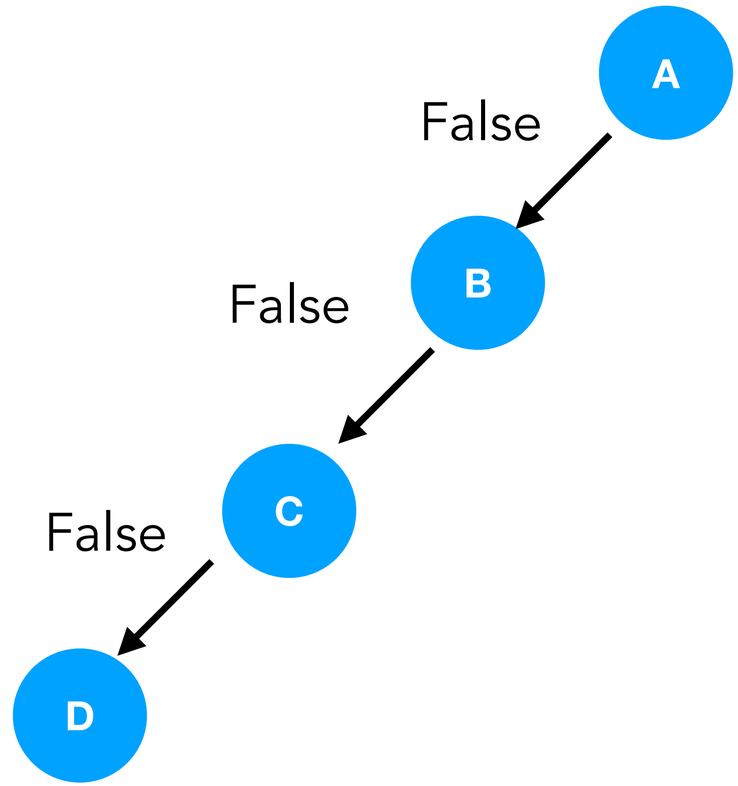
A = False

B = False

C = False

D = ???

Looking only at A is “fine” but still leaves things unfinished:
Let’s keep going—we guess that B and C are also False



Worked Example: DPLL—Unit Propagation

Any clause which has *a single unassigned literal* is called a **unit clause**

Intuitively, a unit clause is a clause which is **forcing the assignment of the single remaining literal**

In this case, *these* clauses are unit

¬A ∨ B ∨ C

A ∨ C ∨ D

A ∨ C ∨ ¬D

A ∨ ¬C ∨ D

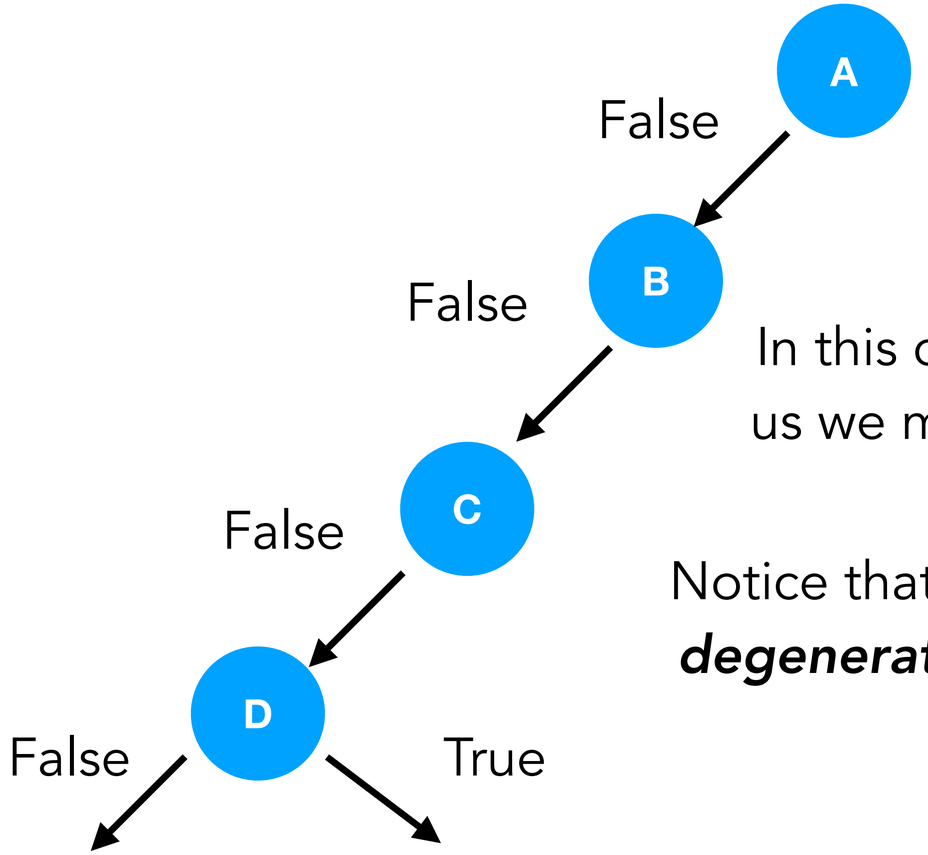
A ∨ ¬C ∨ ¬D

¬B ∨ ¬C ∨ D

¬A ∨ B ∨ ¬C

¬A ∨ ¬B ∨ C

A = False
B = False
C = False
D = ???



In this case, unit propagation tells us we must assign **both** D and ¬D

Notice that unit propagation is a ***degenerate form of resolution***

Worked Example: DPLL—Conflicts

In this case, *these* clauses are unit

¬A ∨ B ∨ C

A ∨ C ∨ D

A ∨ C ∨ ¬D

A ∨ ¬C ∨ D

A ∨ ¬C ∨ ¬D

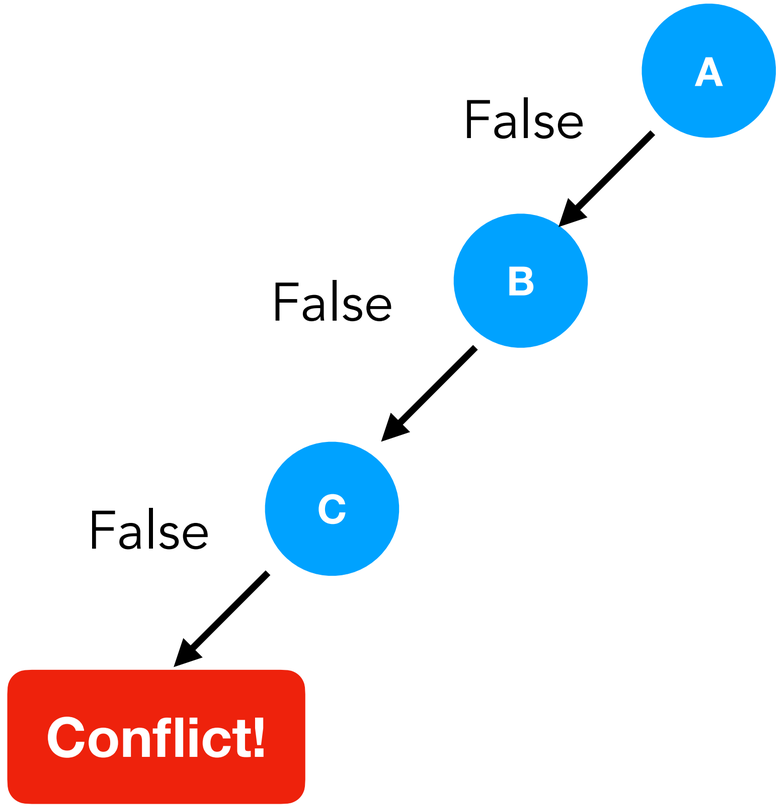
¬B ∨ ¬C ∨ D

¬A ∨ B ∨ ¬C

¬A ∨ ¬B ∨ C

A = False
B = False
C = False
D = ???

We can now observe a **conflict**: unit propagation reveals that both D and ¬D must hold along this branch



Worked Example: DPLL—Implication Graph

In this case, *these* clauses are unit

¬A ∨ B ∨ C

A ∨ C ∨ D

A ∨ C ∨ ¬D

A ∨ ¬C ∨ D

A ∨ ¬C ∨ ¬D

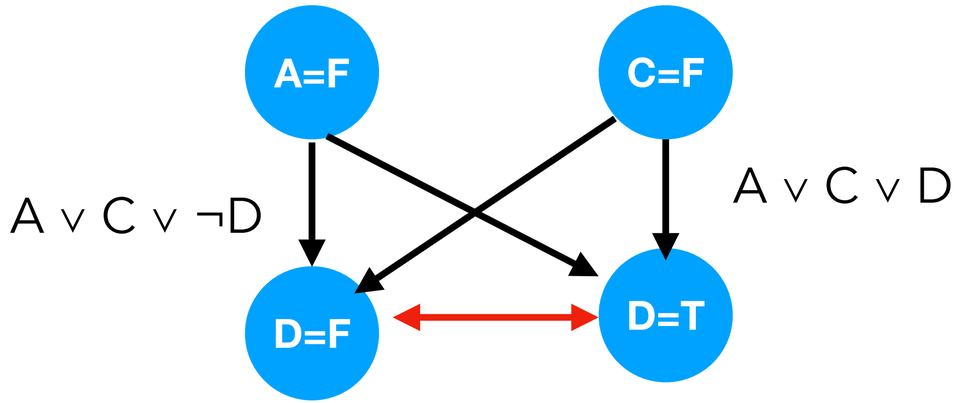
¬B ∨ ¬C ∨ D

¬A ∨ B ∨ C

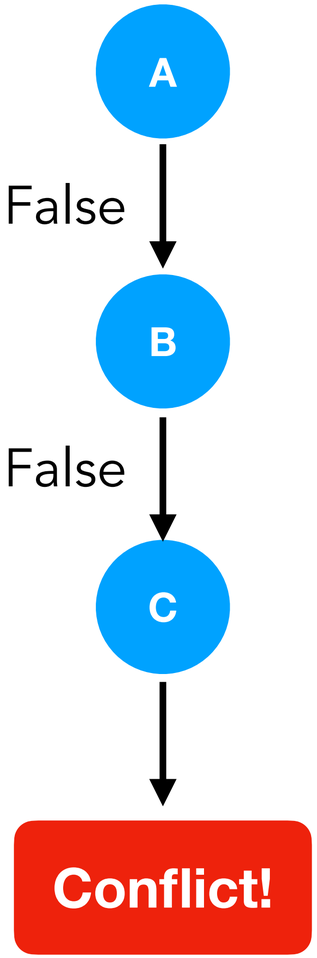
¬A ∨ ¬B ∨ C

A = False
B = False
C = False
D = ???

We can assemble a **(forced) implication graph**, which allows us to record which clauses forced unit propagation to build assignments:



Conflict!



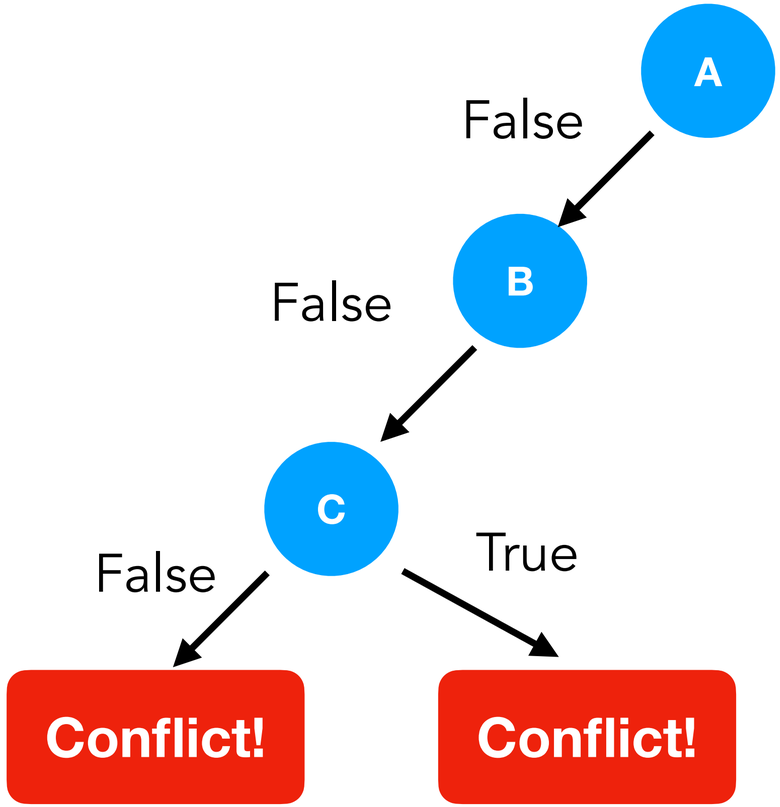
Worked Example: DPLL—Backtracking

Now, *these* clauses are unit

¬A ∨ B ∨ C
A ∨ C ∨ D
A ∨ C ∨ ¬D
A ∨ ¬C ∨ D
A ∨ ¬C ∨ ¬D
¬B ∨ ¬C ∨ D
¬A ∨ B ∨ ¬C
¬A ∨ ¬B ∨ C

A = False
B = False
C = True
D = ???

Once we get to a conflict, we backtrack. In this case we show chronological backtracking back to try C=True



Before each decision, we must repeatedly apply unit propagation—now again, we get a conflict!

Worked Example: DPLL—Keep backtracking

Now, *these* clauses are unit

¬A ∨ B ∨ C

A ∨ C ∨ D

A ∨ C ∨ ¬D

A ∨ ¬C ∨ D

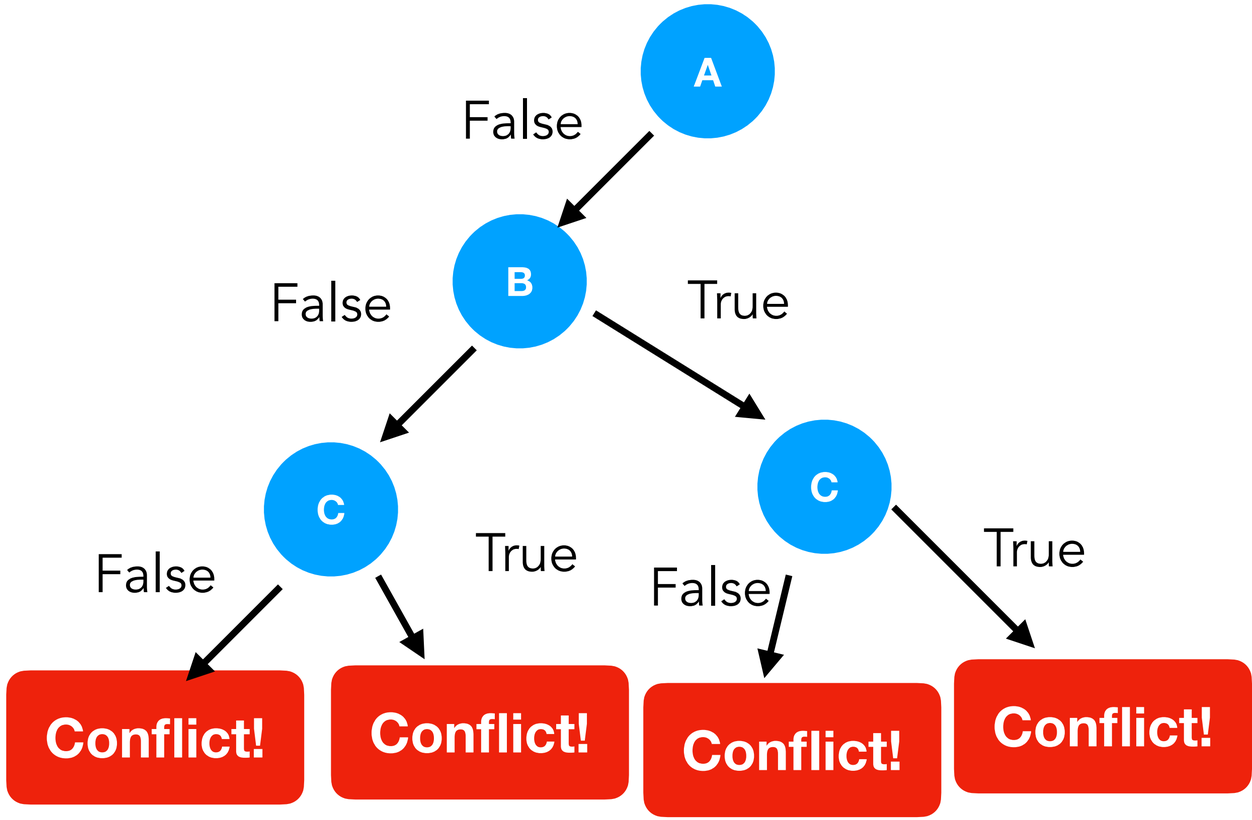
A ∨ ¬C ∨ ¬D

¬B ∨ ¬C ∨ D

¬A ∨ B ∨ ¬C

¬A ∨ ¬B ∨ C

A = False
B = True
C = True
D = ???



So does guessing C = True

Yet again, we get a conflict!

Worked Example: DPLL—Even *more* backtracking…

We backtrack all the way up to guessing A=True, and then guess B=False

These unit clauses conflict!

¬A ∨ B ∨ C

A ∨ C ∨ D

A ∨ C ∨ ¬D

A ∨ ¬C ∨ D

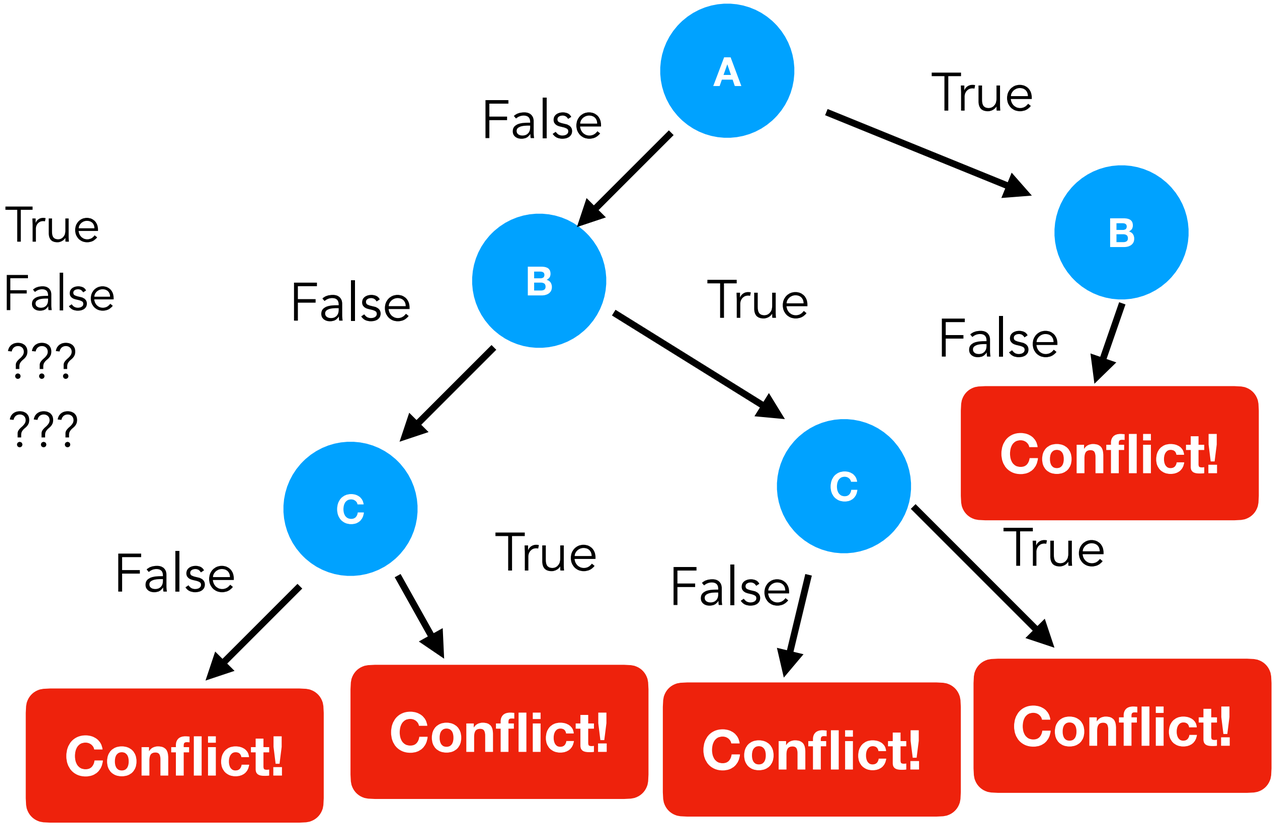
A ∨ ¬C ∨ ¬D

¬B ∨ ¬C ∨ D

¬A ∨ B ∨ ¬C

¬A ∨ ¬B ∨ C

A = True
B = False
C = ???
D = ???



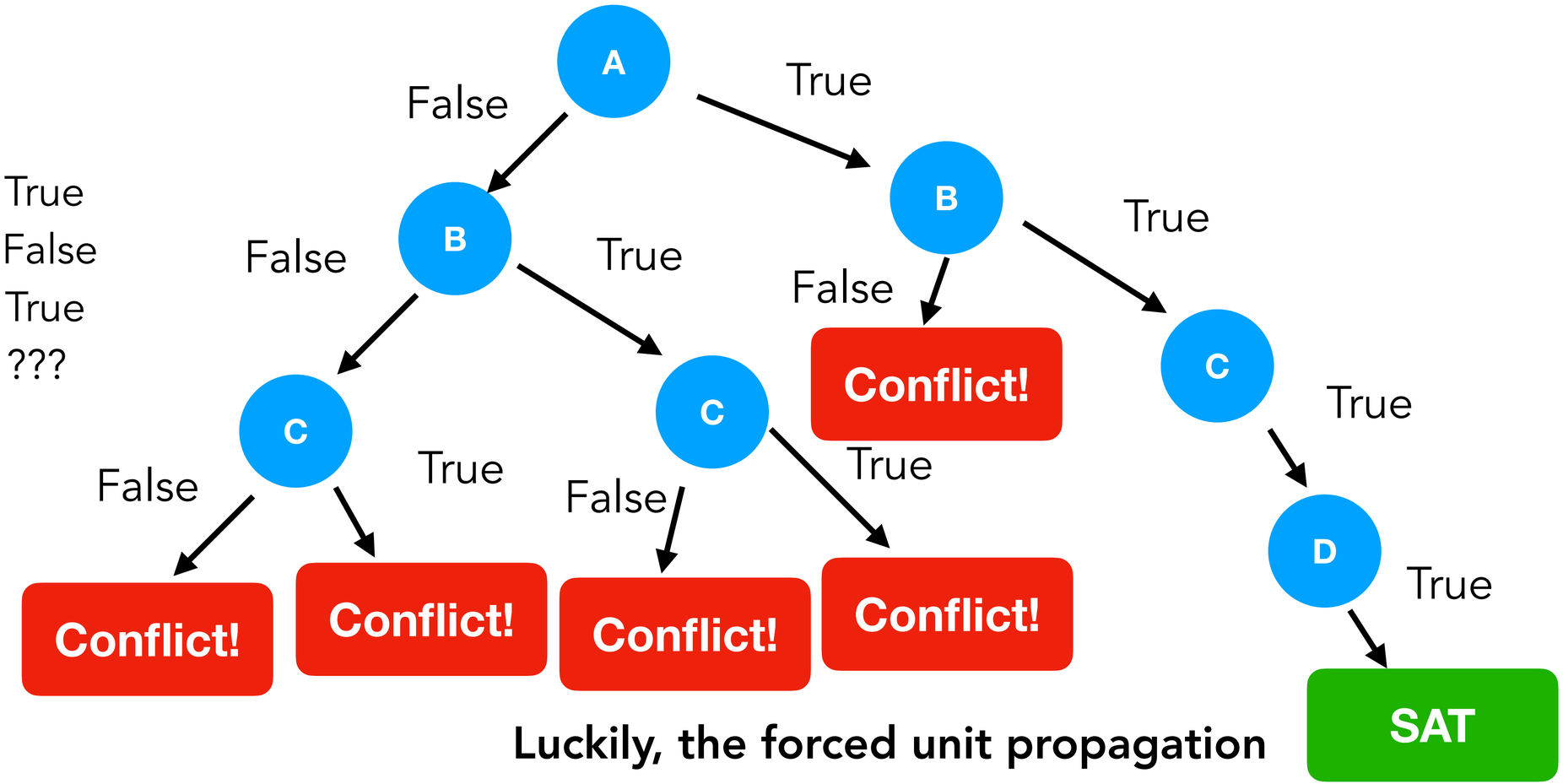
Still a conflict!

Worked Example: DPLL—Success!

The forced assignment of C=True via unit propagation leads to the discovery of a **new** unit clause

¬A ∨ B ∨ C
A ∨ C ∨ D
A ∨ C ∨ ¬D
A ∨ ¬C ∨ D
A ∨ ¬C ∨ ¬D
¬B ∨ ¬C ∨ D
¬A ∨ B ∨ ¬C
¬A ∨ ¬B ∨ C

A = True
B = False
C = True
D = ???



Luckily, the forced unit propagation D=True leads us to success!

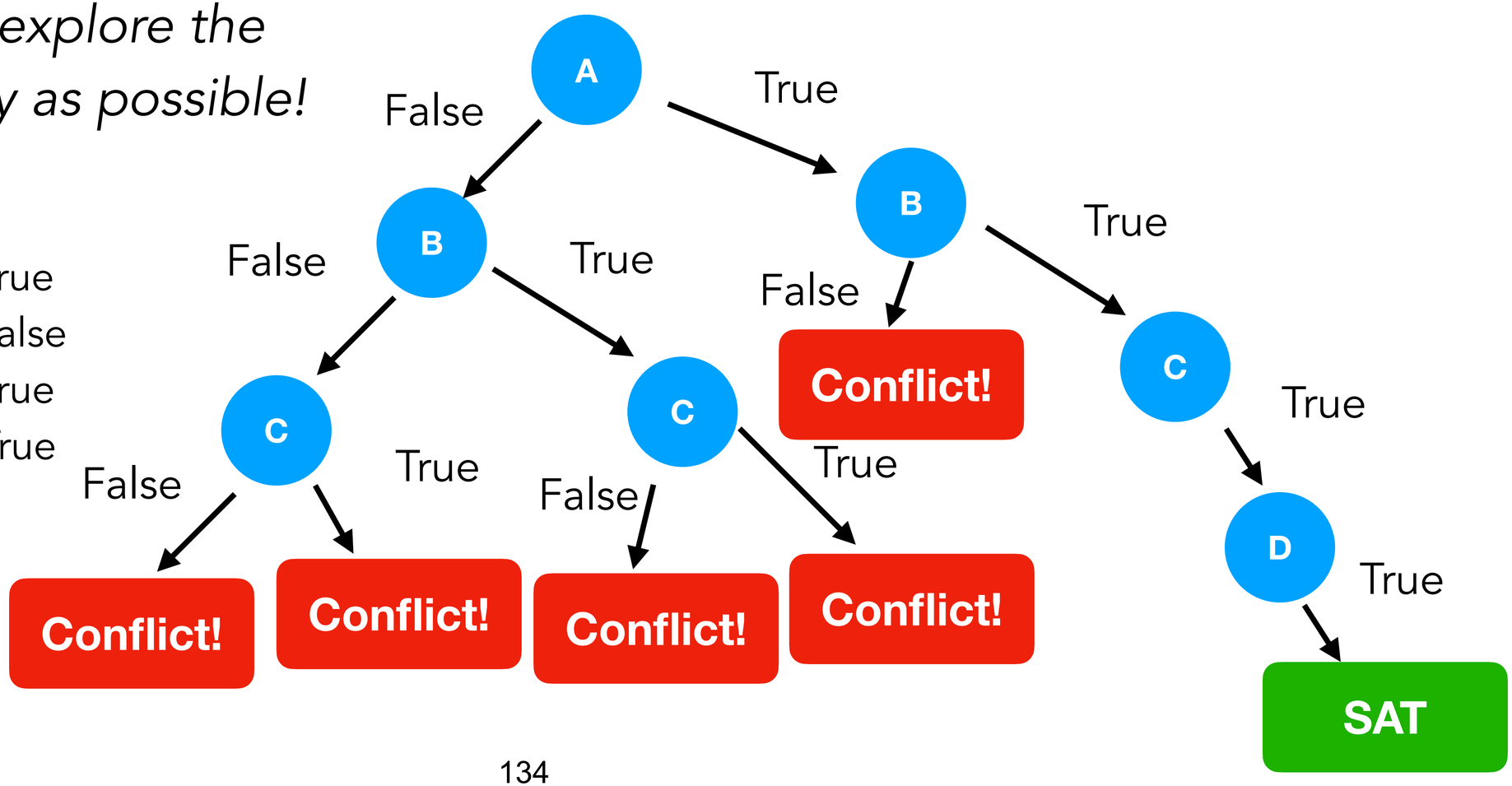
In sum, the DPLL algorithm says to iteratively:

- Apply unit propagation to assignments forced by unit clauses
- If unit propagation leads to conflict, backtrack chronologically (back to most recent guess)
- Decide a variable after all unit clauses taken care of—this is the search phase

The goal is to explore the tree as efficiently as possible!

¬A ∨ B ∨ C
A ∨ C ∨ D
A ∨ C ∨ ¬D
A ∨ ¬C ∨ D
A ∨ ¬C ∨ ¬D
¬B ∨ ¬C ∨ D
¬A ∨ B ∨ ¬C
¬A ∨ ¬B ∨ C

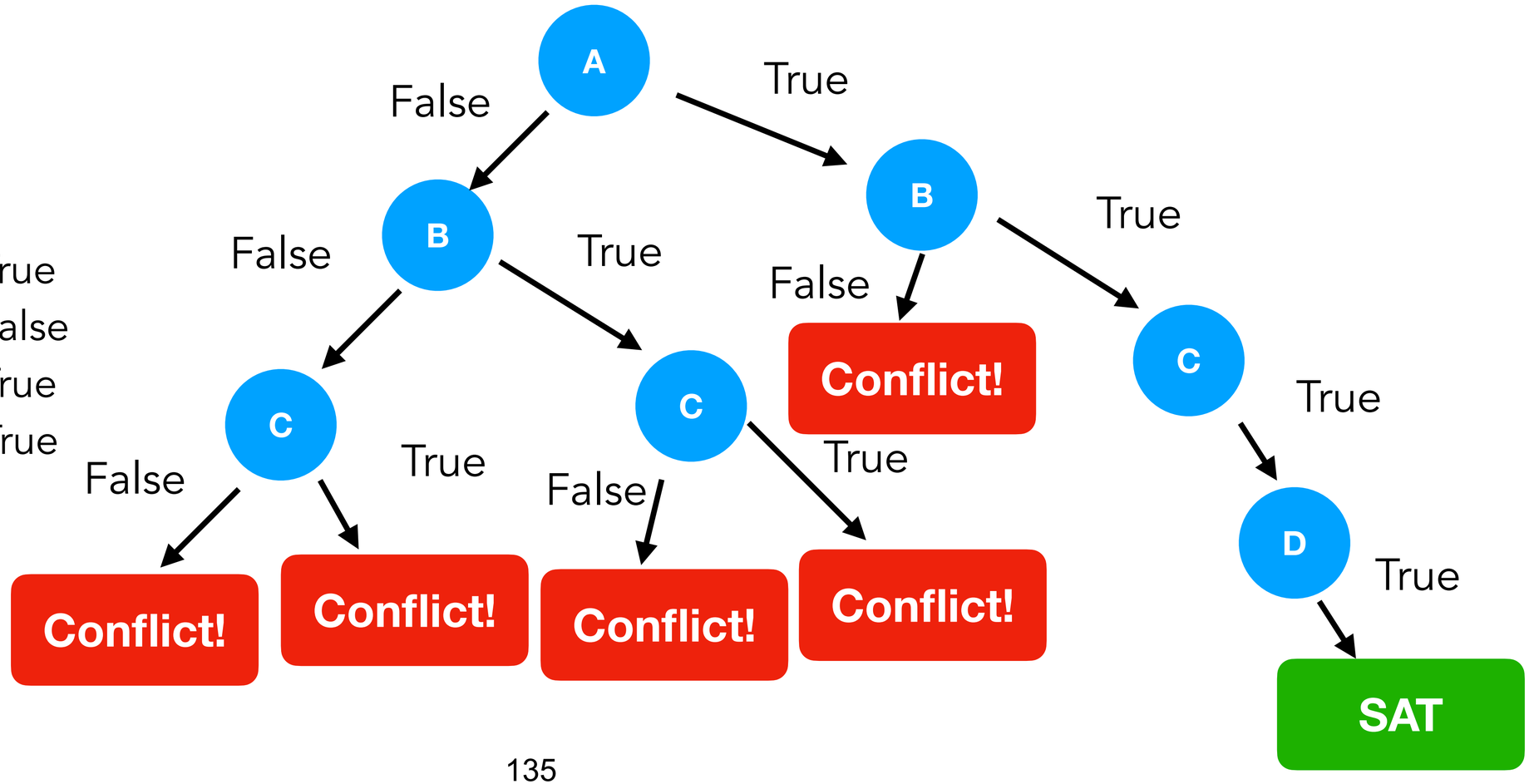
A = True
B = False
C = True
D = True



- Notice that the first choice led us to do a lot of redundant work
- Why didn't we just pick A = True?
- In general: picking variables optimally is tantamount to the halting problem—no general-purpose algorithm exists
- Next week, we'll discuss a **better algorithm** (CDCL) which analyzes conflicts to learn derived clauses that help cut off the search space based on clauses learned on-the-fly

¬A ∨ B ∨ C
A ∨ C ∨ D
A ∨ C ∨ ¬D
A ∨ ¬C ∨ D
A ∨ ¬C ∨ ¬D
¬B ∨ ¬C ∨ D
¬A ∨ B ∨ ¬C
¬A ∨ ¬B ∨ C

A = True
B = False
C = True
D = True



DPLL Algorithm (Wikipedia's definition)

```
function DPLL(Φ)
    // unit propagation:
    while there is a unit clause {l} in Φ do
        Φ ← unit-propagate(l, Φ);
    // pure literal elimination:
    while there is a literal l that occurs pure in Φ do
        Φ ← pure-literal-assign(l, Φ);
    // stopping conditions:
    if Φ is empty then
        return true; // SAT
    if Φ contains an empty clause then
        return false; // UNSAT
    // DPLL procedure:
    l ← choose-literal(Φ);
    return DPLL(Φ ∧ {l}) or DPLL(Φ ∧ {¬l});
```

Pure Literals

Literals are **pure** when they are both (a) unassigned at the current point in the search and (b) they only occur in a single polarity (i.e., only A or only ¬A) in the formula.

Pure variables may simply be discarded—assigning them as either True or False is fine, and so they do not force decisions.

DPLL *alternates* between **inferring immediate consequences** (i.e., "saturation") and **guessing** (i.e., "decision")

```
function DPLL(Φ)
    // unit propagation:
    while there is a unit clause {l} in Φ do
        Φ ← unit-propagate(l, Φ);
    // pure literal elimination:
    while there is a literal l that occurs pure in Φ do
        Φ ← pure-literal-assign(l, Φ);
    // stopping conditions:
    if Φ is empty then
        return true; // SAT
    if Φ contains an empty clause then
        return false; // UNSAT
    // DPLL procedure:
    l ← choose-literal(Φ);
    return DPLL(Φ ∧ {l}) or DPLL(Φ ∧ {¬l});
```

Saturation Phase

Decision Phase

Aside: Horn clauses and Datalog

Horn clauses are clauses with **at most one** positive (i.e., non-negated) literal

P ← Q, R (Logic programming style)
or Q ∧ R → P (implication style)
or (definition of →, DeMorgan…) ¬(Q ∧ R) ∨ P ≡ ¬Q ∨ ¬R ∨ P

Datalog also allows facts: atomically known propositions (which can be interpreted as → P, i.e., nothing needed to infer P)

P ← Q, R
Q ← K
K
G
R ← J
J ←G

Query: P?

Datalog is **easier** to decide than SAT—the degenerate nature of Horn clauses means that we never have to *guess*. Datalog can be solved via saturation, **without** the need for guessing or backtracking. It's complexity lies in PSPACE (<< than k-SAT!)

DPLL Algorithm

Input — set of clauses ϕ in CNF
Output — True (SAT) or False (UNSAT)

DPLL(ϕ):
  Forever:
    While there exist any unit clauses $\{l\} \in \phi$:
      ϕ := unit_propagate(ϕ,l)
    If ϕ contains no more clauses: **return** True
    Elif ϕ contains any empty clauses: **return** False
    Else:
      Choose a literal l which is unassigned in ϕ
      Return DPLL($\Phi \wedge \{l\}$) $\vee$ DPLL($\Phi \wedge \{\neg l\}$)

A few remarks:

DPLL(ϕ):
  Forever:
    // Want to avoid scanning over all of ϕ
    While there exist any unit clauses {l} ∈ ϕ:
      // Unit Propagation needs to be fast
      ϕ := unit_propagate(ϕ,l)
    If ϕ contains no more clauses: **return** True
    Elif ϕ contains any empty clauses: **return** False
    Else:
      // How do we pick variables?
      Choose a literal l which is unassigned in ϕ
      Return DPLL(Φ ∧ {l}) ∨ DPLL(Φ ∧ {¬l})

Project 1 – DPLL/SAT

CIS700 — Fall 2023

Kris Micinski

This project has a fairly short specification.

Your job is to build a SAT solver.
- Your SAT solver should be invoked on the command line. You may write in any language you want: Python, C++, Racket, Haskell, etc…
- Your SAT solver should take a single argument, which is a file in DIMACS CNF input format. The format is very simple—it should be possible to parse it by reading each line and using a string split operation
- Your program should write an output to the console of either:
  - (a) the string “UNSAT” or (b) a satisfying assignment to the formula.

DIMACS CNF Input Format

Read the following short guide:

https://jix.github.io/varisat/manual/0.2.0/formats/dimacs.html

Example (from that site): encode $(x \vee y \vee \neg z) \wedge (\neg y \vee z)$ as

```
p cnf 3 2
1 2 -3 0
-2 3 0
```

First line specifies number of propositions (sequentially numbered), followed by number of clauses

Each subsequent line gives a clause, consisting of whitespace-separated literals (either l, positive l or -l, not l), ending in 0 (end of line sentinel)

Projects are due on the 28th, anywhere on earth (so, 7am on the 29th in syracuse).

You may work in groups of up to three. If you are relatively more advanced, please try to work in a group of students which is less confident.

If you are less interested in dedicating time to the class, it is acceptable to do the baseline of simply enumerating the SAT instances and checking all of them.

If you are feeling you seriously want to learn this material implement either (a) DPLL or (c) CDCL

Submissions and Grading

Please email me submissions, kkmicins@syr.edu, CCing all of your group mates.

I will test your submissions with a variety of DIMACS inputs, e.g., the ones from this page. https://people.sc.fsu.edu/~jburkardt/data/cnf/cnf.html

Please tell me how to invoke and run your program when you email me. Ensure that it can run on either a Mac or Linux machine (I have both of these)—give me sources and instructions to build your project.

Grading / Rubric

- I will give you a 95% if your project is a good-faith attempt that is a working solution for small CNF instances. If you get occasional hiccups, I am fine with that. If you get a substantial number of instances incorrect (i.e., wrong model/ unsat) I will consider taking off some points. Ensure that your implementation is correct—I will grade whether or not you have tests for your project. I expect **some** testing infrastructure, even if it is relatively basic—I am not expecting production quality, but a convincing argument that it was constructed with an eye to correctness
- I will give you +5% if you implement more than enumerate-and-check, i.e., if you implement DPLL or something similar
- I will give you +5% *bonus* (so, 105%) if you implement CDCL (which we will discuss next week) as well. And I will be very impressed.

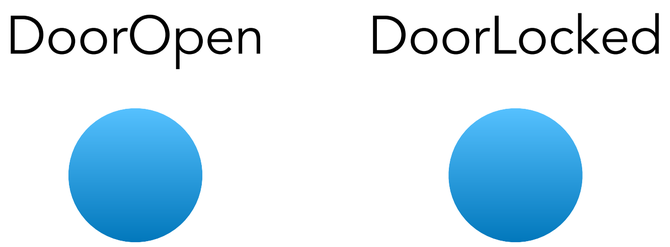
Model Checking & Büchi Automata

Modern Symbolic AI & Automated Reasoning

Kris Micinski (Fall 2023)

In today's class, we'll talk about how we could verify *trace properties* of finite-state systems.

The new notion we'll focus on is the notion of a *trace semantics* of a system. In our case, we'll start very simple: transition systems which are labeled with atomic propositions.



Horn Clauses and Datalog

CIS700 — Fall 2023

Kris Micinski

Today, we'll talk about how to operationalize the rules from last class as a specific programming paradigm: logic programming

Review: Resolution

The resolution rule tells us how to infer new knowledge from preexisting knowledge

$$\frac{P \vee \ldots \vee Q \qquad \neg Q \vee R \ldots \vee}{P \vee \ldots R \vee \ldots}$$

If we derive $\perp$, we know the original formula is tantamount to $\perp$

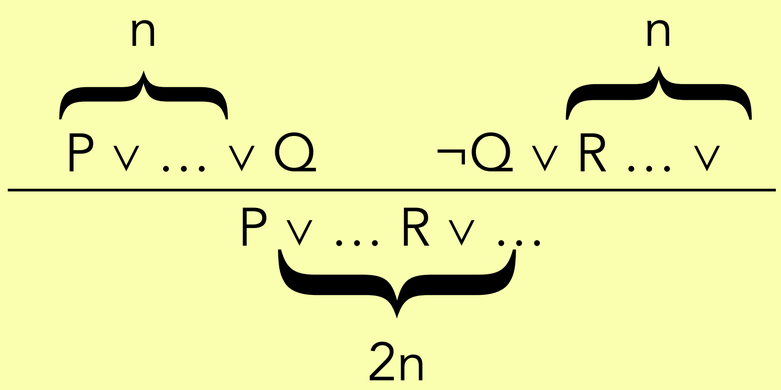
We can view resolution as giving us the **transitive closure of our current knowledge base to explicate latent implications**

The Problem with Resolution

Resolution may or may not be helpful—it may produce new clauses which are not useful

Churning on unproductive work can be very costly

Resolution-based solvers must judiciously select how to apply resolution



In future lectures, we'll see how DPLL, CDCL, and related systems apply resolution intelligently (but heuristically) to scale SAT solving to formulas with tens of thousands of variables and millions of clauses.

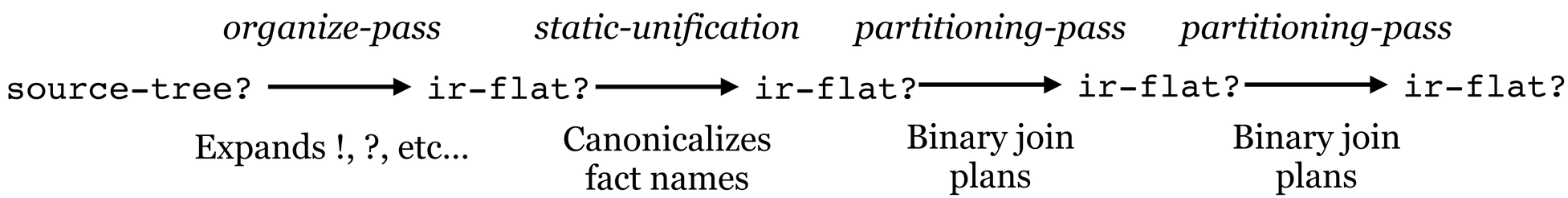
Today, we'll focus on a simpler logic programming language based on a restricted form of clauses

Horn Clauses

A horn clause is a clause with at most one positive (i.e., not negated) literal

$\neg B_0 \vee \ldots \vee \neg B_n \vee H$ or, equivalently… $H \leftarrow B_0 \wedge \ldots B_n$

H is the “head” of the clause, and the B_ns are the “body”
"If everything in the body is true, the head must be true”


organize-pass
static-unification
partitioning-pass
partitioning-pass
source-tree?
ir-flat?
ir-flat?
ir-flat?
ir-flat?
Expands !, ?, etc...
Canonicalizes fact names
Binary join plans
Binary join plans

Datalog

Horn clauses allow **chain forward** reasoning: if the body is true, then the head must be true

The language **Datalog** implements chain forward Horn clauses over a universe of atoms; in this lecture we'll look at Datalog, its foundations and applications, and its implementation

Forward vs. Backward Reasoning

Logical systems often include **inference rules**, which define schemes which define how to deduce new knowledge from preexisting knowledge

We'll talk more about these rules soon when we see natural deduction. For now, let's look at the only rule we've seen so far

Forward vs. Backward Reasoning

Logical systems often include **inference rules**, which define schemes which define how to deduce new knowledge from preexisting knowledge

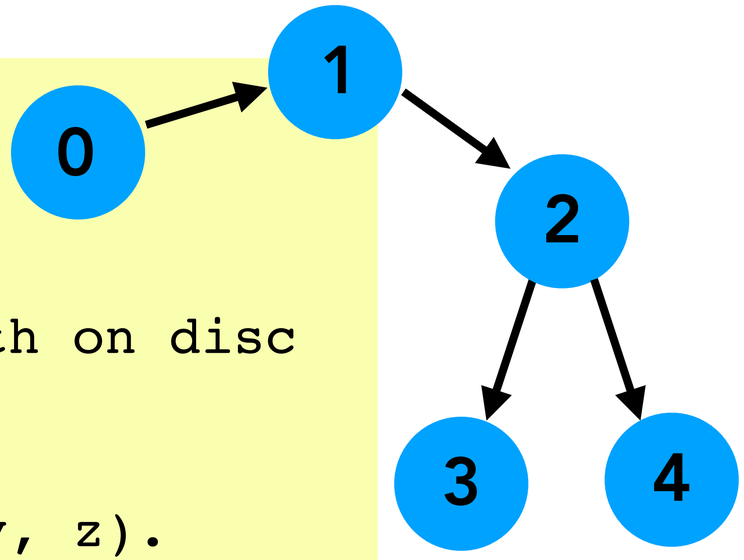
We'll talk more about these rules soon when we see natural deduction. For now, let's look at the only rule we've seen so far

Example: Transitive Closure in Soufflé

```
// Transitive Closure
.decl edge(x:number, y:number)
.decl path(x:number, y:number)
.output path // materializes path on disc

path(x, y) :- edge(x, y).
path(x, z) :- path(x, y), edge(y, z).
```

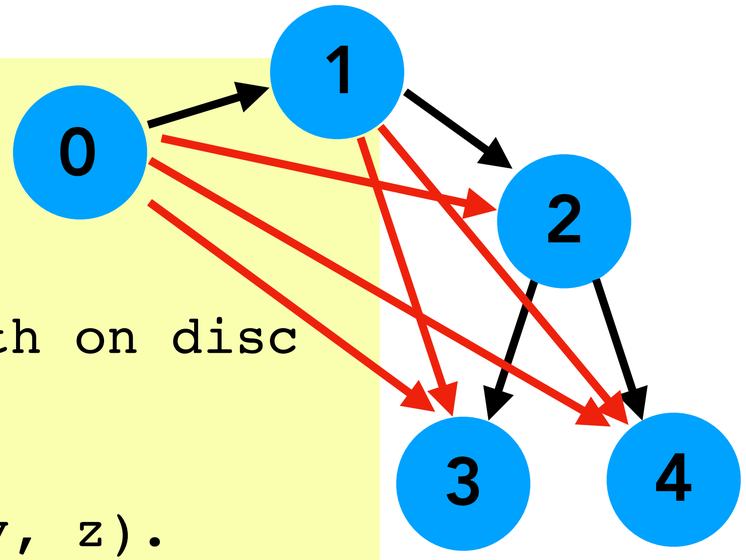
Input: Extensional DataBase (EDB)

```
// Transitive Closure
.decl edge(x:number, y:number)
.decl path(x:number, y:number)
.output path // materializes path on disc

path(x, y) :- edge(x, y).
path(x, z) :- path(x, y), edge(y, z).

// Extensional DataBase (EDB)
edge(0,1). edge(1,2). edge(2,3). edge(2,4).
```



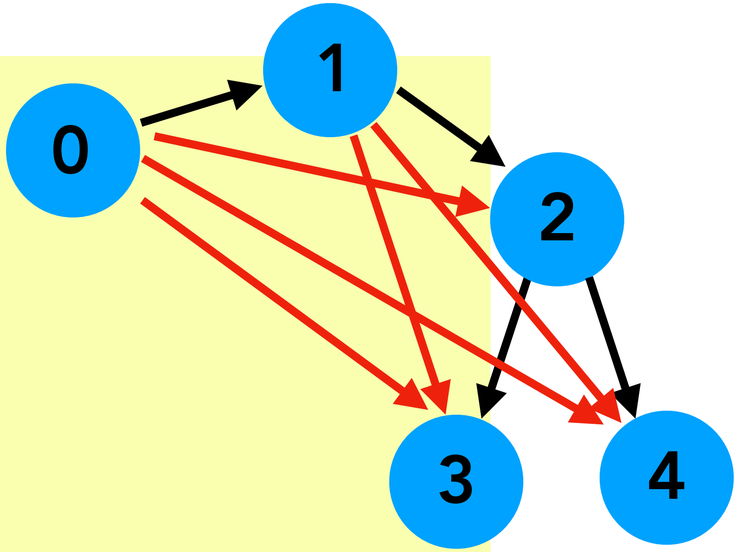
Computation **materializes** the result

```
// Transitive Closure
.decl edge(x:number, y:number)
.decl path(x:number, y:number)
.output path // materializes path on disc

path(x, y) :- edge(x, y).
path(x, z) :- path(x, y), edge(y, z).

// Extensional DataBase (EDB)
edge(0,1). edge(1,2). edge(2,3). edge(2,4).
```



Let’s run it and see

```
kmicinski % souffle tc.dl
kmicinski % cat path.csv
0	1
0	2
0	3
0	4
1	2
1	3
1	4
2	3
2	4
```



Challenge: Triangle Counting, etc…

Write a Soufflé program which takes an input edge of the same form as before. You should output triples

How does this generalize to k-clique (k > 3)?

What is (worst-case) runtime complexity of k-clique, increasing with k? (Hint: k-clique is NP complete!)

Conjunction in the rule heads

A conjunction in the head is technically disallowed:

$$H_0 \wedge H_1 \leftarrow B_0 \wedge \ldots B_n$$

But this is only superficial: we can simply refactor this into two rules

$$H_0 \leftarrow B_0 \wedge \ldots B_n$$

$$H_1 \leftarrow B_0 \wedge \ldots B_n$$

Disjunction in rule heads

Horn clauses allow **chain forward** reasoning: if the body is true, then the head must be true

Notice that this rules out (a) negation in the body and (b) disjunction in the head; consider the alternative:

$$H_0 \vee H_1 \leftarrow B_0 \wedge \ldots B_n \qquad H_0 \vee H_1 \vee \neg B_0 \vee \ldots \vee \neg B_n$$

Here, when we know the body is true, we know that either $H_0 \vee H_1$ is true—this means we need to consider *both* possibilities

This extension is known as disjunctive Datalog; it enables encoding NP-complete problems (TSP, …)

We will soon talk about the design of modern SAT solvers, which may primarily be viewed as search algorithms;

These solvers repeatedly

Hilbert Systems, Sequent Calculus

CIS700 — Fall 2023

Kris Micinski

Congruence Closure, Union-Find, and Equality

CIS700 — Fall 2023

Kris Micinski

DPLL and SAT Solver Design

CIS700 — Fall 2023

Kris Micinski

In this lecture, we'll start to look at automated theorem provers for classical logics

These solvers accept a goal ϕ (a formula) as input; we assume ϕ contains some number of propositional variables Vars

The solvers return **True** when ϕ is a validity, and **False** when ϕ is not a validity

Propositional logic is decidable (to see: construct a truth table)

Which of these are valid?

- $(P \Longrightarrow (Q \wedge R)) \Longrightarrow (P \Longrightarrow (R \wedge \neg \neg Q))$
- $(P \vee R) \wedge Q \Leftrightarrow (R \vee Q) \wedge P$
- $P \wedge \neg(Q \vee R) \Leftrightarrow \neg R \wedge \neg Q \wedge P$

Which of these are valid?

- $(P \Longrightarrow (Q \wedge R)) \Longrightarrow (P \Longrightarrow (R \wedge \neg \neg Q))$ **yes**
- $(P \vee R) \wedge Q \Leftrightarrow (R \vee Q) \wedge P$ **no** (P false, R/Q true)
- $P \wedge \neg(Q \vee R) \Leftrightarrow \neg R \wedge \neg Q \wedge P$ **yes**

How do we **prove** that a formula is valid?

Idea: truth tables. Decidability is

Interpretations

A formula is just a *statement*. To speak of a statement's veracity, we need to rigorously define the notion of a **true statement**

There are deeply differing perspectives on this:

- A statement is true precisely when I can materialize a proof for it
  - This is the **constructive** view, every statement demands *evidence* represented as data
- Truth based on a **model**
  - Model tells me what is true / false

Propositional Interpretations

An interpretation

We can represent the set of natural numbers as an inductively defined set in the following way:

The set of natural numbers includes the number 0, and for any natural number n, the number n+1

How / why might we formalize this:
How? Structurally-inductive algebraic datatype including two objects:
O : nat
∀x : nat, S(x) : nat
Why? Can implement in a computer (using language features, inductive types, pattern matching, etc…)

In Lean, we

Natural Deduction / IPL

CIS700 — Fall 2023

Kris Micinski

Last lecture: a formula is **valid** if it is “true” (i.e., *holds*) in every interpretation (assignment to free propositional variables)

This is the so-called “classical” view: everything is true, or it is false. There is a universe of discourse (here the propositional variables) giving their definitions

In this lecture, we will look at a different perspective on logic: **constructive** logic

In a constructive proof system, we can only say statements are "true" when we have a symbolic proof for them

Why do *we* (this course) care about constructivity? Gives us a symbolic representation of knowledge, which we may implement via a data structure in a computer program

We do this by giving natural deduction rules for the various connectives

Here is our most basic rule: “if we assume P, then we may prove P.”

The rule has an **antecedent** (set of things which must be true, in this case nothing) above the line

$$\frac{}{P \vdash P}\ \text{Assumption}$$

And a **consequent** below the line

Our **logical statements** will be of the form $\phi_0 \vdash \phi_1$, read “assuming ϕ_0, we may prove ϕ_1.”

To actually write a proof, we instantiate the rule; the rule is schematic on the propositional variable P

$$\frac{}{Q \vdash Q}\ \text{Assumption} \qquad \frac{}{R \vdash R}\ \text{Assumption}$$

"Assuming Q, we can prove Q"

"Assuming R, we can prove R"

Introduction and Elimination Rules

For each logical connective, we define two types of forms: **introduction** forms (how do I *build* an $\wedge$, …) and **elimination** forms (how do I *use* an $\wedge$, …)

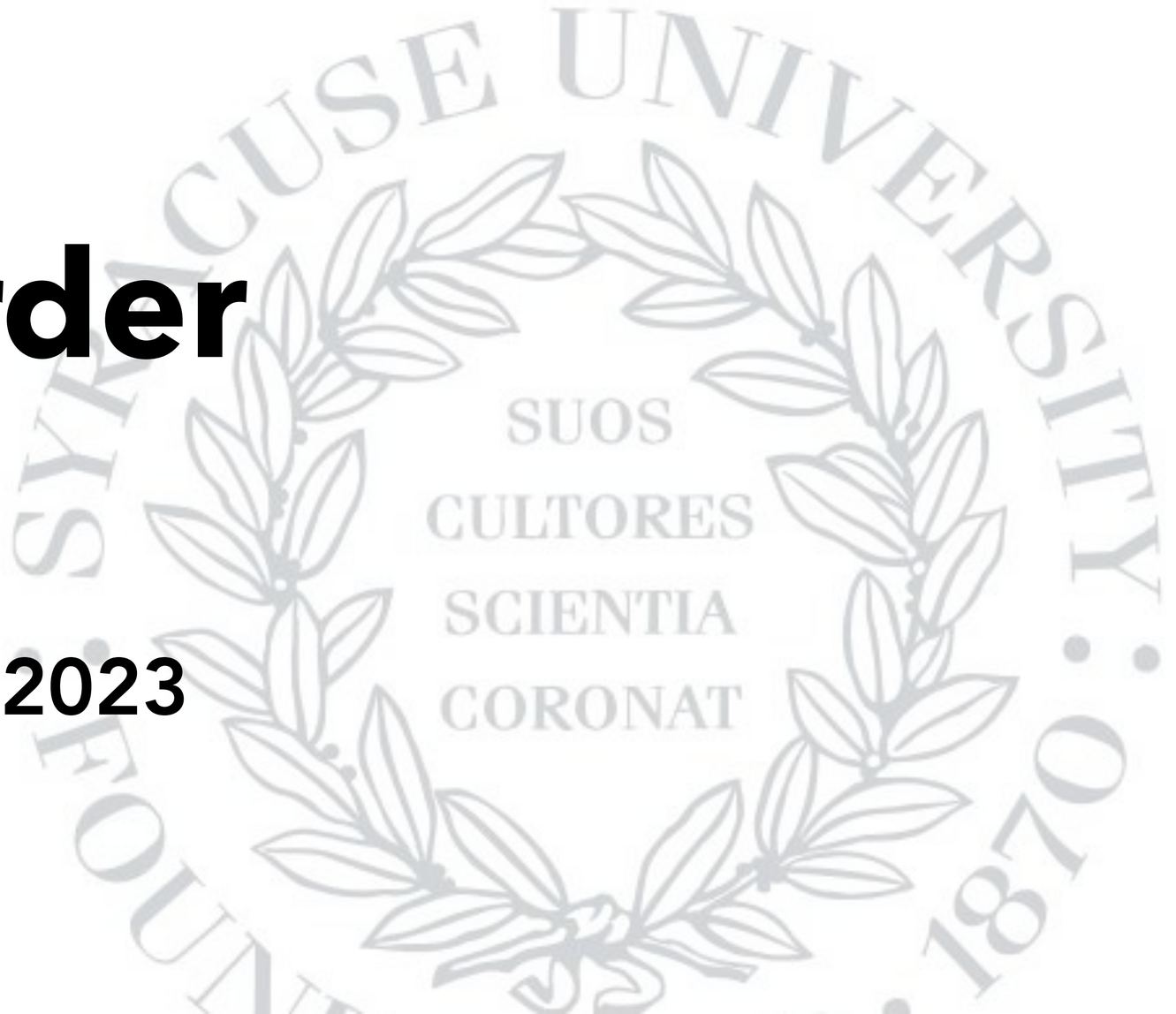
Assumption

$\vdash P \wedge Q$

First-Order Logic

CIS700 — Fall 2023

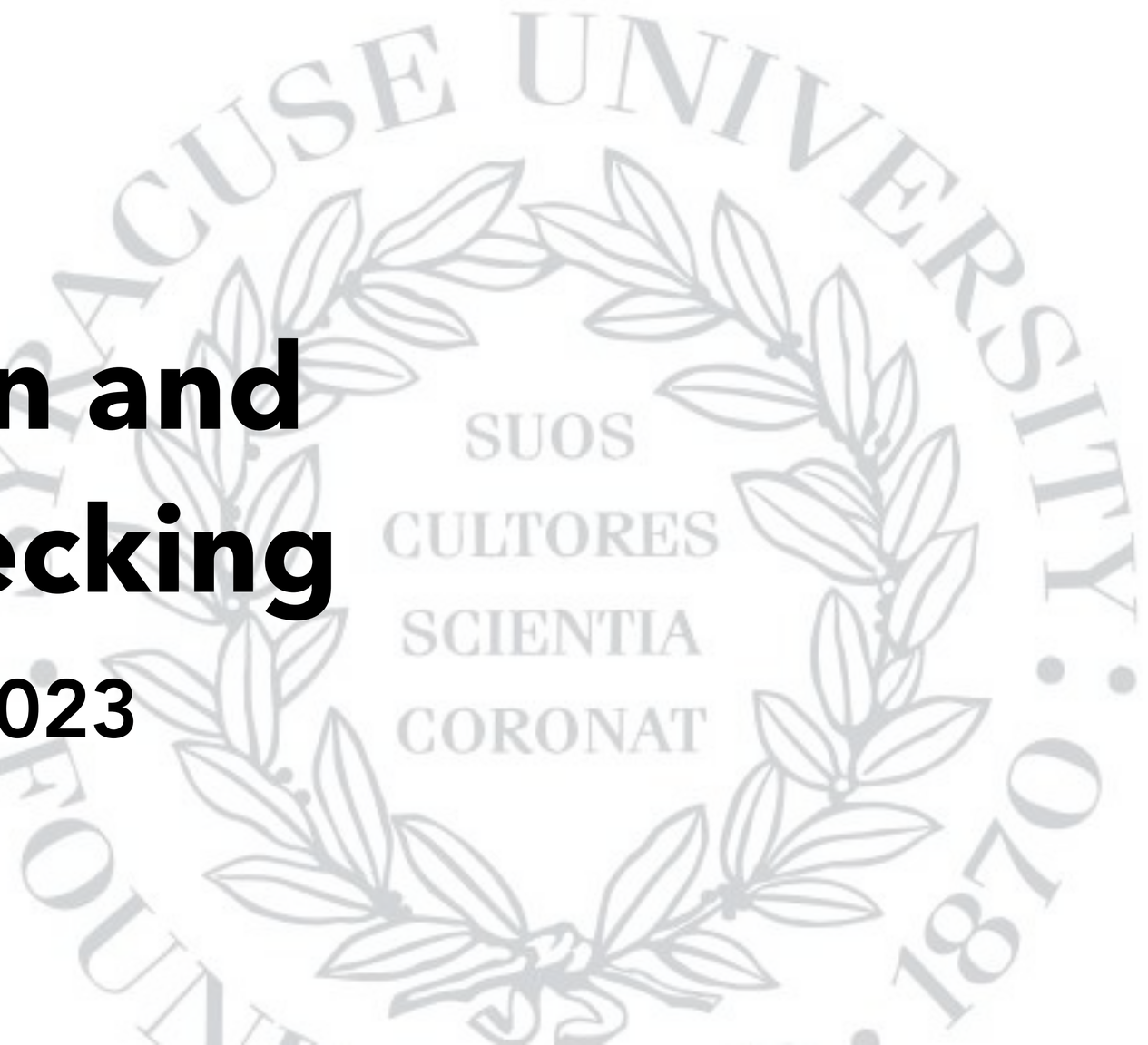
Kris Micinski



Predicate Abstraction and Model Checking

CIS700 — Fall 2023

Kris Micinski



Satisfiability Modulo Theories

CIS700 — Fall 2023

Kris Micinski

First-Order Theorem Provers

CIS700 — Fall 2023

Kris Micinski

Monotonic Aggregation

CIS700 — Fall 2023

Kris Micinski