AI-Powered
Theorem Prover Guidance

Zarathustra Amadeus Goertzel

EuroProofNet Summer School on AI for Reasoning and Processing of Mathematics

Kutaisi International University, June 2024

Outline

1. I will introduce learning in general terms.

2. Theorem-proving (TP): formal mathematical libraries, the superposition calculus, and saturation-based theorem provers.

3. Combining AI and TP, focusing on my work and some other gems.

Caveats

- My knowledge is imperfect; please forgive mistakes and feel free to pitch in if I miss an important detail.
    - Fortunately, perfect knowledge isn't needed to contribute to state-of-the-art research 😎.

- I'll focus on my recent PhD research and some related methods, which form a significant part of a general overview of the field.
- I know a lot about AI for Theorem Proving to varying fuzzy degrees that may not make it into the slides – feel free to ask!
    - I may be able to at least point you to sources 🤓.

- I may broadly point to some rabbit holes w/o going down them: 🕳🐇

Learning

- Learning is the process of acquiring knowledge or skill with experience.

- Given:
  - A domain of possible experiences, D
  - A parametrized agent, π_θ
  - A task defined by a performance measure, $\phi(\Theta,D) \rightarrow R$

  - A learning algorithm L adjusts the agent’s parameters θ such that the expected performance over D increases.

- I adapted this from Schmidhuber and Schaul’s Scholarpedia article: Metalearning.

Learning

- Learning is the process of acquiring knowledge or skill with experience.

- Given:
  - A domain of possible experiences, D
  - A parametrized agent, π_θ
  - A task defined by a performance measure, $\phi(\Theta, D) \rightarrow R$

  - A learning algorithm L adjusts the agent's parameters θ such that the expected performance over D increases.

Note:

1. Sometimes measuring loss makes more sense.
2. Non-statistical notions can also be formulated.

Learning – Examples

- Is an *associative array* learning?
- The data are pairs of *keys* and *values*: D = K x V.
- The *task* is defined by $\phi(\theta,(k,v)) = 1$ if $\pi_\theta(k) == v$ and 0 otherwise.
- Prior to building the array, the performance is likely 0 on a training set $T \subset D$.
- After building the array on T, the performance is perfect on T.

- The expected performance on D should be non-zero….

Learning – Examples

- Is an *associative array* learning?
- The data are pairs of *keys* and *values*: D = K x V.
- The *task* is defined by $\phi(\theta,(k,v)) = 1$ if $\pi_\theta(k) == v$ and 0 otherwise.
- Prior to building the array, the performance is likely 0 on a training set $T \subset D$.
- After building the array on T, the performance is perfect on T.
- The expected performance on D should be non-zero….

- So it is learning by this simple definition. But we want more!

Learning – Examples

- Is an *associative array* learning?
- The data are pairs of *keys* and *values*: $D = K \times V$.
- The task is defined by $\phi(\theta,(k,v)) = 1$ if $\pi_\theta(k) == v$ and 0 otherwise.
- Prior to building the array, the performance is likely 0 on a training set $T \subset D$.
- After building the array on T, the performance is perfect on T.
- The expected performance on D should be non-zero….

- Pros: time saved due to indexing?
- Cons:
  1. Zero generalization to any $d \in D$ but not T.
  2. No size compression.

Learning – Examples

- Let T = {(6,1), (5,0), (99,0), (100,1), (42,1), (23,0),...}
- An array could become very large….
- The hypothesis that for all pairs, (k,v), v = even(k) fits the dataset perfectly where `even = lambda x : ((x+1) % 2)`.
- This solution may also generalize to new data points.

- Symbolic regression is an approach to learning that could achieve this.
- *Program synthesis* could, too.

Learning – Examples

- Let T = {(6,1), (5,0), (99,0), (100,1), (42,1), (23,0),...}
- The hypothesis that for all pairs, (k,v), v = even(k) fits the dataset perfectly where `even = lambda x : ((x+1) % 2)`.
- Symbolic regression is an approach to learning that could achieve this.
- *Program synthesis* could, too.

- Pros:
  1. Time and space-wise efficient solutions.
  2. Able to capture the properties of the data distribution
- Cons:
  1. huge search space
  2. data may not have a lossless compact solution

Learning – Examples

1. This perspective on learning works for clustering algorithms, too.
2. The agent learns some centroids.
3. The performance (loss) measure measures the distance from the nearest centroid.
4. Learning aims to minimize the expected loss.

What is not learning?

- If building a key-indexed array is bare-minimal “learning”, what doesn’t count?
- Copying a list of data to another directory in the computer?
- Deleting the dataset (probably won’t improve the performance)?
- … 🤔

Do We Need Symbolic Learning?

1. Consider arithmetic, which LLMs are not that good at [*]:
    - Even if they *can* do “1+1=2”, is that efficient?

Do We Need Symbolic Learning?

1. Consider arithmetic, which LLMs are not that good at [*]:
   - Even if they *can* do “1+1=2”, is that efficient?
     – No.

- We have efficient hardware and software algorithms for arithmetic.

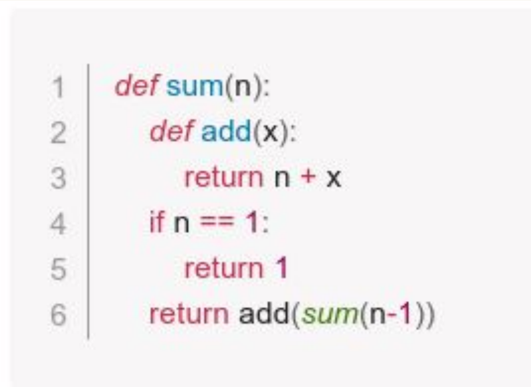
- At best, LLM-based systems should delegate arithmetic tasks to other systems, perhaps via code.

Do We Need Symbolic Learning?

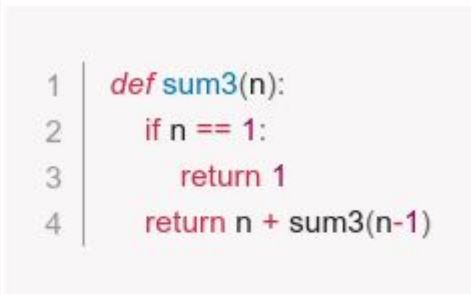
2. Even among symbolic solutions, the approach matters.

- I like the example of Gauss’ technique to sum numbers from 1 to 100.

- Using “+” is *3.4x faster* than locally defining an “add x” function.



```
def sum(n):
    def add(x):
        return n + x
    if n == 1:
        return 1
    return add(sum(n-1))
```



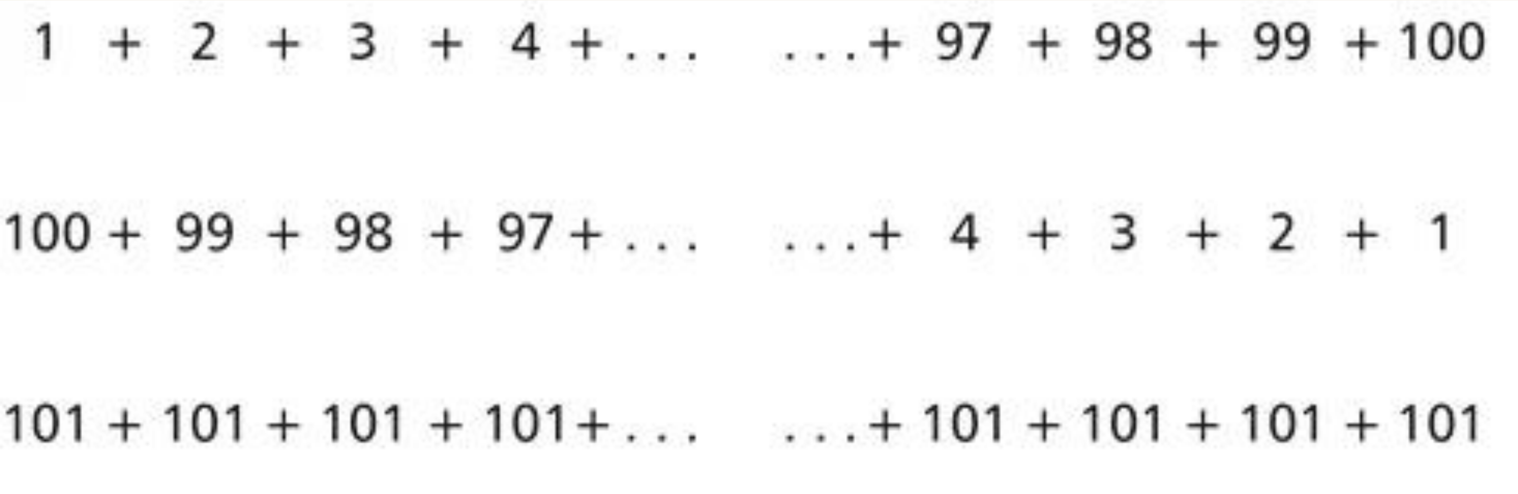
```
def sum3(n):
    if n == 1:
        return 1
    return n + sum3(n-1)
```

Do We Need Symbolic Learning?

2. Even among symbolic solutions, the approach matters.

- I like the example of Gauss’ technique to sum numbers from 1 to 100.

- Gauss’ nice trick leads to a symbolic solution… is this better?



$$1 + 2 + 3 + 4 + \ldots \quad \ldots + 97 + 98 + 99 + 100$$

$$100 + 99 + 98 + 97 + \ldots \quad \ldots + 4 + 3 + 2 + 1$$

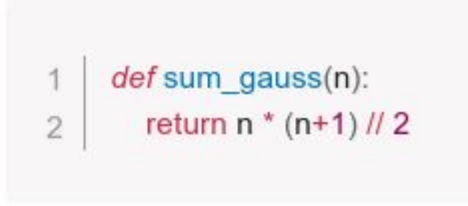
$$101 + 101 + 101 + 101 + \ldots \quad \ldots + 101 + 101 + 101 + 101$$

Do We Need Symbolic Learning?

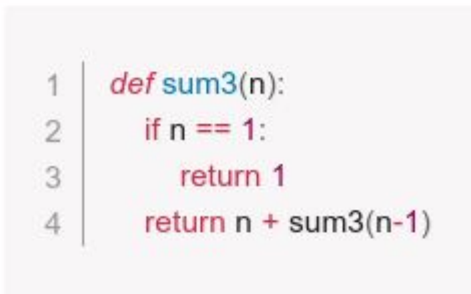
2. Even among symbolic solutions, the approach matters.

- I like the example of Gauss’ technique to sum numbers from 1 to 100.

- Gauss’ nice trick leads to a symbolic solution… is this better?
- Yes, it’s 9.3x faster!
  - (But only 2.3x faster than the for loop.)



```python
def sum_gauss(n):
    return n * (n+1) // 2
```



```python
def sum3(n):
    if n == 1:
        return 1
    return n + sum3(n-1)
```

Automated Theorem Proving

- General problem solving.
- Helps mathematicians.
- Helps with the formalization of mathematics.
- Hardware and software verification.
- Automated reasoning for general AI systems, hard sciences, etc.
    - Provides the grounding in truthfulness that LLM-based systems lack.

Automated Theorem Proving

- General problem solving.
- Helps mathematicians.
- Helps with the formalization of mathematics.
- Hardware and software verification.
- Automated reasoning for general AI systems, hard sciences, etc.
    - Provides the grounding in truthfulness that LLM-based systems lack.

- When an ATP (automated theorem prover) is (refutation) complete, it becomes a *universal search procedure*, technically general AI.

Proofs as Programs 🕳️🐇

- Curry-Howard Correspondence: constructive proofs of mathematical propositions can be regarded as programs of the type of the proved proposition.
- Ex: for a list L, len(L) == len(reverse(L)). A proof could involve programs to reverse the list, measure the length, and check for equality.

Proofs as Programs 🕳️🐇

- Curry-Howard Correspondence: constructive proofs of mathematical propositions can be regarded as programs of the type of the proved proposition.
- Ex: for a list L, len(L) == len(reverse(L)). A proof could involve programs to reverse the list, measure the length, and check for equality.

- Takeaway: theorem proving is related to program synthesis.
  - Both of which are highly general methods.

Automated Theorem Proving:

Historical successes

- Formal Verification of Large Proofs:
  - Formal Proof of the Kepler Conjecture (2014 – Hales – 20k lemmas)
  - Formal Proof of the Feit-Thompson Theorem (2 books, 2012 – Gonthier)
- Software Verification:
  - Verification of compilers (CompCert) ,
  - Verification of OS microkernels (seL4), HW chips (Intel)
  - Verification of cryptographic protocols (Amazon), etc.
- Novel Human-Machine Proofs:
  - Proof of the Weak AIM conjecture in loop theory (2021 – Kinyon and Veroff)
  - Proof of Robbins conjecture that Robbins algebras are Boolean algebras (1996 – McCune)

Automated Theorem Provers

- Automated Theorem Provers (ATPs) are programs that try to determine if:
  - *A conjecture* ***C*** *is a logical consequence of a set of axioms* ***Ax****.*
- Usually use *brute-force search* within a *logical calculus* (e.g., *resolution, superposition, connection/tableaux, inst-gen,...*).
- Systems: E, Vampire, CVC5, Prover9, Z3, iProver, leanCoP…
- Theoretically complete yet practically run into *combinatorial explosions*
  - Thus the need for learning and AI guidance!

Is understanding logic important for applying AI to guide ATPs?

- Imagine training neural networks to work with visual images without knowing much about pictures and sight.
  - When something goes wrong, making sense of the errors and debugging may be difficult.

- Likewise, when working with formal mathematics, some understanding of the various logical languages used can help in various subtle ways.

What is Logic?

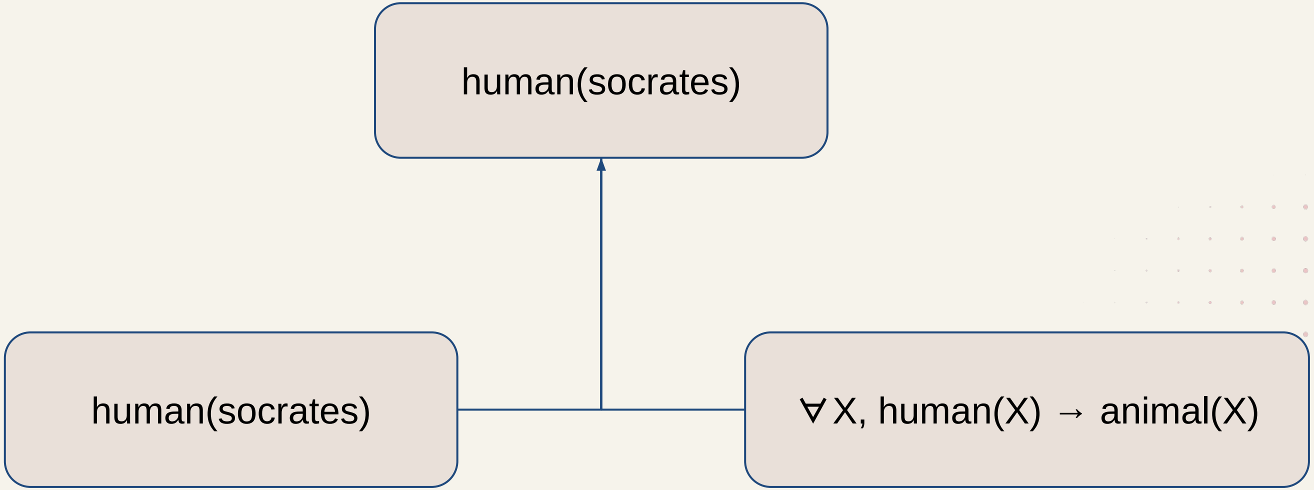
- Philosophically, *logic* aims to concretely and *syntactically* represent human thought and sound reasoning.

What is Logic?

- Philosophically, *logic* aims to concretely and *syntactically* represent rational thought and sound reasoning.
- Q: when does is a statement *justified* by other statements?
    - E.g., “Socrates was an animal because he was human and all humans are animals.”

What is Logic?

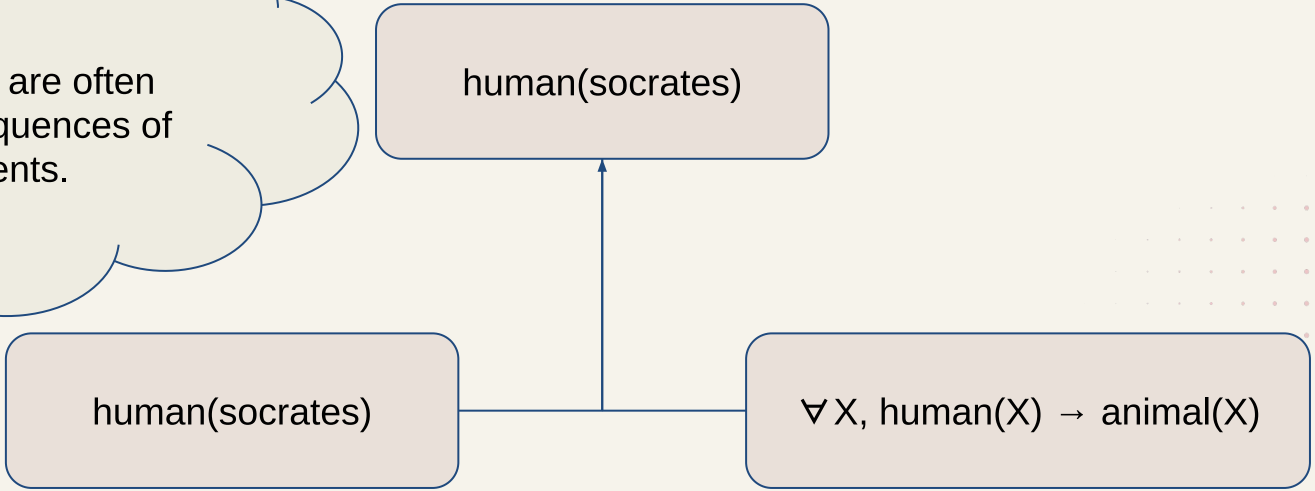
- Q: when does is a statement *justified* by other statements?
  - E.g., “Socrates was an animal because he was human and all humans are animals.”
- Proofs are *directed acyclic graphs* where each *node* is a *statement* and *(multi)edges* are *logical inferences.*



What is Logic?

- Q: when does is a statement *justified* by other statements?
  - E.g., “Socrates was an animal because he was human and all humans are animals.”
- Proofs are *directed acyclic graphs* where each *node* is a *statement* and *(multi)edges* are *logical inferences.*

<u>Note</u>: Proofs are often viewed as sequences of statements.



human(socrates)

human(socrates)

∀X, human(X) → animal(X)

What is Logic? – Negative Example

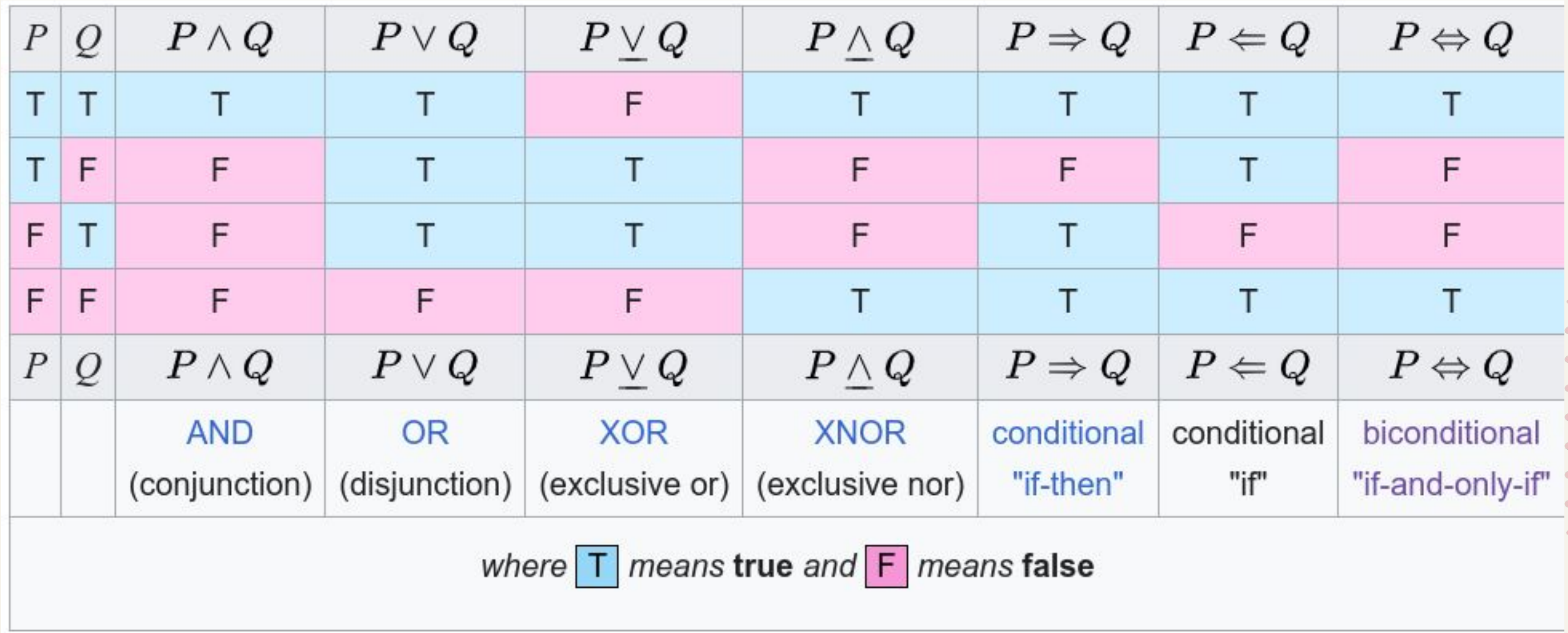
- Consider the Affirming the Disjunct logical fallacy:
  1. I am at home or I am in the city.
  2. I am at home.
  3. Therefore, I am not in the city.
- Both A and B can be true when (A or B) is true, therefore (not B) cannot be inferred from (A and (A or B)).

Kinds of Logic

- Propositional Logic:
    - Deals with *propositions* that represent the truth value of statements (*true or false*).
    - Standard operators: ∧ (“and”), ∨ (“or”), ¬ (“not”), → (“implication”), and ↔ (“equivalence”).
        - (Both NAND and NOR are *functionally complete* alone.)
    - An *interpretation* assigns meaning to the symbols.
        - *Truth-value semantics* allows one to analyze truth tables over propositional variables.

Kinds of Logic

- Propositional Logic Truth Table:

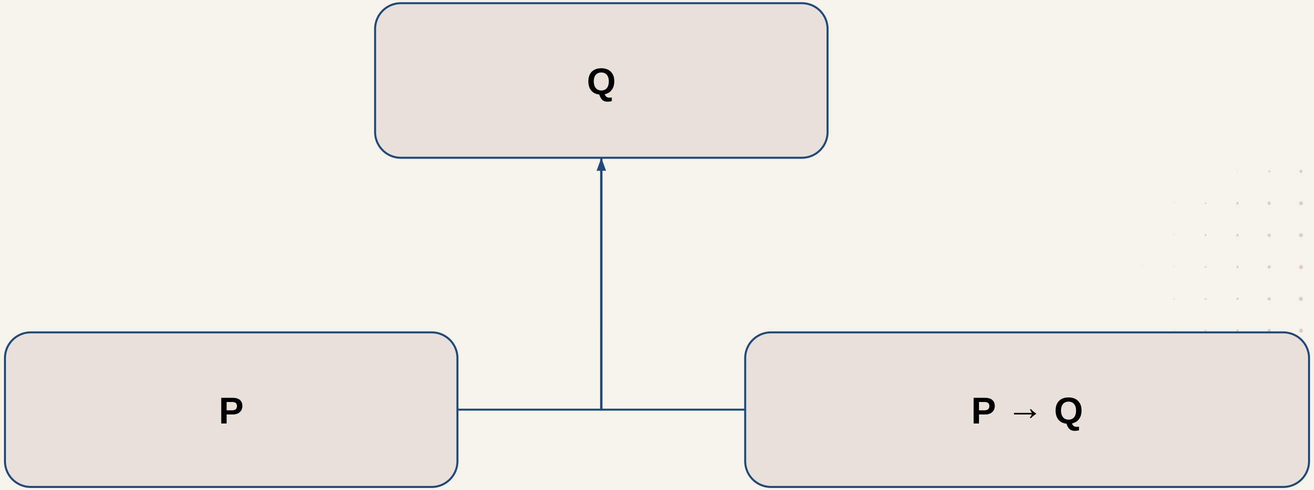


P	Q	$P \wedge Q$	$P \vee Q$	$P \mathbin{\underline{\vee}} Q$	$P \mathbin{\underline{\wedge}} Q$	$P \Rightarrow Q$	$P \Leftarrow Q$	$P \Leftrightarrow Q$
T	T	T	T	F	T	T	T	T
T	F	F	T	T	F	F	T	F
F	T	F	T	T	F	T	F	F
F	F	F	F	F	T	T	T	T
P	Q	$P \wedge Q$	$P \vee Q$	$P \mathbin{\underline{\vee}} Q$	$P \mathbin{\underline{\wedge}} Q$	$P \Rightarrow Q$	$P \Leftarrow Q$	$P \Leftrightarrow Q$
		AND (conjunction)	OR (disjunction)	XOR (exclusive or)	XNOR (exclusive nor)	conditional "if-then"	conditional "if"	biconditional "if-and-only-if"

where T *means* **true** *and* F *means* **false**

Kinds of Logic

- Propositional Logic Example:
  - Let **P** :- “Socrates is a man”
  - Let **Q** :- “Socrates is an animal”
  - Then, **P** → **Q** :- “*If* ‘Socrates is a man’, *then* ‘Socrates is an animal’”



Kinds of Logic

- Propositional Logic:
  - Deals with *propositions* that represent the truth value of statements (*true or false*).
  - Standard operators: ∧ (“and”), ∨ (“or”), ¬ (“not”), → (“implication”), and ↔ (“equivalence”).
    - (Both NAND and NOR are *functionally complete* alone.)
  - An *interpretation* assigns meaning to the symbols.
    - *Truth-value semantics* allows one to analyze truth tables over propositional variables.
  - A *formula* is *satisfiable* if there exists an interpretation such that it’s true.
    - E.g., **P** → **Q** is satisfiable (with “False -> True”) and **P** ∧ ¬**P** is not satisfiable.
  - An interpretation satisfying a (set of) formula(s) is called a *model* of the formula(s).

- Deciding the satisfiability of propositional formulas is the domain of SAT solvers.
  - In general, it’s an *NP-Complete* problem.

Kinds of Logic

- First-Order Logic (FOL):
  - Allows one to employ *predicates* and to *quantify* over variables with regard to a *universe of discourse*.
    - Thus we can now say “all humans are animals”, “∀X, human(X) → animal(X)”.
    - Predicates allow one to discuss properties of entities and relations among entities.
  - The *existential quantification* $\exists_x P(x)$ is intended to be true if and only if there is an existential witness *a* in *U* such that p(a) is true.
  - The *universal quantification* $\forall_x P(x)$ is intended to be true if and only if P(a) is true for every element *a* in *U* .
  - We also get *function symbols* to be interpreted as n-ary functions from U^n to U.
  - *Terms* are inductively defined as *variables* or functions whose arguments are terms.
  - *Atomic formulas* consist of predicates applied to terms (including equality as a predicate).
  - *Well-formed formulas* are inductively made of atomic formulas, quantified formulas, and compound formulas constructed with propositional connectives (and, or, not…).

Kinds of Logic

- First-Order Logic (FOL):
  - Interpretations require additional *structure M*:
    - A *variable assignment function μ* for free variables.
    - A *universe U* containing the entities discussed.
    - A *signature* of the language, specifying the *constant, function, and predicate* symbols that can be used (and their arities).
    - An *interpretation* assigning to each constant an element of *U*, a function U^n to U for every n-ary function symbol, and a subset of U^n for every n-ary predicate symbol.

  - In a given structure, a formula *F* is:
    - *satisfiable* if there's a variable assignment such that F is true.
    - *valid* if it is true in all variable assignments.
    - *logically valid* if it is true in all interpretations.
  - *M* is a model for a theory T (a set of sentences) if it satisfies every sentence in T.
    - Where a *sentence* is a formula with no free variables.

Kinds of Logic

- First-Order Logic is pretty well-behaved.
- Two kinds of logical consequence (implication):
    - Syntactic consequence: $Ax \vdash F$ – F can be proved from Ax (in a formal system)
    - Semantic consequence: $A \vDash F$ – every model of A is also a model of F.
- Soundness: syntactic implies semantic consequence.
- Semantic Completeness: semantic implies syntactic consequence.

- There are multiple formal proof systems for first-order logic that are both sound and complete.
    - 🥳🎉

- Satisfiability is fully *undecidable*, but…
- Logical validity is *semi-decidable* (like the *halting problem*): if a sentence is valid, then a decision procedure can determine this; but if it is not valid, the procedure may not terminate.

    – This is great news for theorem-proving!

Kinds of Logic

- Many-Sorted First-Order Logic: we can extend FOL with multiple universes over which variable can be quantified.

Kinds of Logic

- Many-Sorted First-Order Logic: we can extend FOL with multiple universes over which variable can be quantified.
- Second-Order Logic: variables can be instantiated with predicates (‘subsets of the universe’) instead of only elements of the universe.

Kinds of Logic

- Many-Sorted First-Order Logic: we can extend FOL with multiple universes over which variable can be quantified.
- Second-Order Logic: variables can be instantiated with predicates (‘subsets of the universe’) instead of only elements of the universe.
- Simple Type Theory, Higher-Order Logic (HOL):
  - Base types: ι for *individuals* and * for *truth values*.
  - Predicates are defined as the type $(\iota \to *)$. Either an element is in a relation or not.
  - Function types, $(\alpha \to \beta)$, are defined inductively where α and β are types.
  - Quantification is restricted by type.

  Interpretation:
  - Let D_α be the set of all values of type α.
  - *Standard models*: function domains, $D_{(\alpha \to \beta)}$, contain all total functions from D_α to D_β.
  - *General models*: function domains may contain non-empty sets of the total functions.
    - With general models, simple type theory is semantically equivalent to first-order logic!
    - (See The Seven Virtues of Simple Type Theory for a good introduction.)

Kinds of Logic

- Intuitionistic/Constructive Logic: classical logic without the law of excluded middle or double negation (¬¬A → A).
    - There are “double negation translations” such that F is provable in classical logic if and only if ¬¬F’ is provable in intuitionistic logic!
- Paraconsistent Logics: aim to develop deductive systems in which the law of explosion is invalid, that is, one cannot derive any proposition from a contradiction.
- Fuzzy Logics: allow truth values to take numerical values between in [0,1].
- Probabilistic Logics: aim to extend crisp logical entailment to uncertain, probabilistic inferences.
- Linear Logics: assign a cost to doing inferences, so that one can reason about resource usage.
- Modal Logics: allow one to reason about diverse modalities or *‘worlds’* with different rules of operation: e.g., *belief, knowledge, possibility, necessity, temporality, and obligation*.

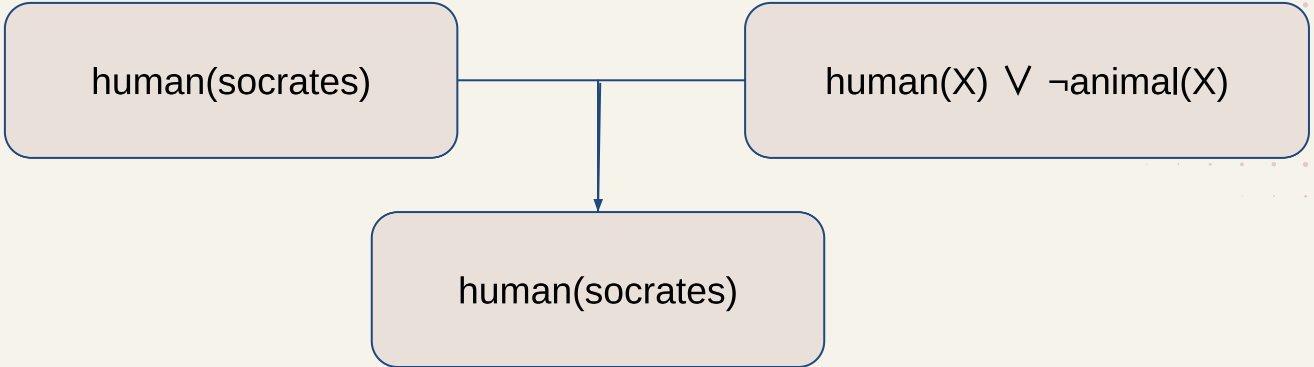
Kinds of Logic

- Intuitionistic/Constructive Logic: classical logic without the law of excluded middle or double negation (¬¬A → A).
    - There are "double negation translations" such that F is provable in classical logic if and only if ¬¬F' is provable in intuitionistic logic!
- Paraconsistent Logics: aim to develop deductive systems in which the law of explosion is invalid, that is, one cannot derive any proposition from a contradiction.
- Fuzzy Logics: allow truth values to take numerical values between in [0,1].
- Probabilistic Logics: aim to extend crisp logical entailment to uncertain, probabilistic inferences.
- Linear Logics: assign a cost to doing inferences, so that one can reason about resource usage.
- Modal Logics: allow one to reason about diverse modalities or *'worlds'* with different rules of operation: e.g., *belief, knowledge, possibility, necessity, temporality, and obligation*.

- I hope that work on AI for theorem proving can extend from FOL to most of these logics (as they may be helpful in dealing with *"the real world"*).

Clausification: Clausal Normal Form

- Most ATPs and SAT solvers work with formulas in *clausal/conjunctive normal form (CNF)*.
    - Atomic formulas (and their negations) are called *literals*.
    - A *clause* is a *disjunction (“or”)* of literals.
    - Quantifies are moved to the outermost scope.
    - Existential quantifiers are replaced with *Skolem symbols* (existential witnesses).
    - The result is *refutationally equivalent*, unsatisfiability is preserved.
    - All variables are implicitly universally quantified.



human(socrates)

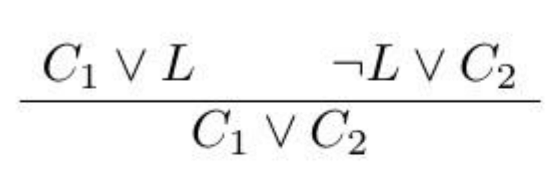
human(X) ∨ ¬animal(X)

human(socrates)

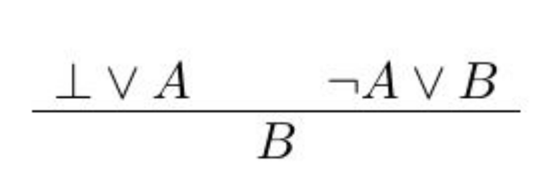
Resolution and Superposition Calculi

- Some of the strongest ATPs use the *superposition calculus*: Zipperposition, E, Vampire,….
- As David showed, they are *refutationally complete*.
  - (And, actually, pretty flexible to work with by hand, too!)

- Note that *modus ponens* (A and A -> B imply B) is a special case of resolution.



$$\frac{C_1 \vee L \qquad \neg L \vee C_2}{C_1 \vee C_2}$$



$$\frac{\bot \vee A \qquad \neg A \vee B}{B}$$

Resolution and Superposition Calculi

- Some of the strongest ATPs use the *superposition calculus*: Zipperposition, E, Vampire,….
- As David showed, they are *refutationally complete*.

- A *substitution* σ is a mapping from variables to terms.
- A *unifier* of terms *t* and *s* is a substitution such that sσ = tσ.
- A *most general unifier (mgu)* is one that can be specialized via substitution into any other unifier.
- We can say that *s matches t* if there is a substitution such that sσ = t.
- For FOL, resolution is done with respect to a mgu: σ = mgu(L1 , L2).

$$\frac{C_1 \vee L_1 \qquad \neg L_2 \vee C_2}{(C_1 \vee C_2)\sigma}$$

Resolution and Superposition Calculi

- Some of the strongest ATPs use the *superposition calculus*: Zipperposition, E, Vampire,….
- As David showed, they are *refutationally complete*.

- A *substitution* σ is a mapping from variables to terms.
- A *unifier* of terms *t* and *s* is a substitution such that sσ = tσ.
- A *most general unifier (mgu)* is one that can be specialized via substitution into any other unifier.
- We can say that *s matches t* if there is a substitution such that sσ = t.
- For FOL, resolution is done with respect to a mgu: σ = mgu(L1 , L2).

$$\frac{C_1 \lor L_1 \qquad \neg L_2 \lor C_2}{(C_1 \lor C_2)\sigma}$$

$$\frac{Man(\text{socrates}) \qquad \neg Man(x) \lor Animal(x)}{Animal(\text{socrates})}\ [\text{socrates}/x]$$

Resolution and Superposition Calculi

- The factoring rule unifies literals within a clause. So with σ = mgu(L1, L2):

$$\frac{C_1 \vee L_1 \vee L_2}{(C_1 \vee L_2)\sigma}$$

Resolution and Superposition Calculi

- The factoring rule unifies literals within a clause. So with σ = mgu(L1, L2):

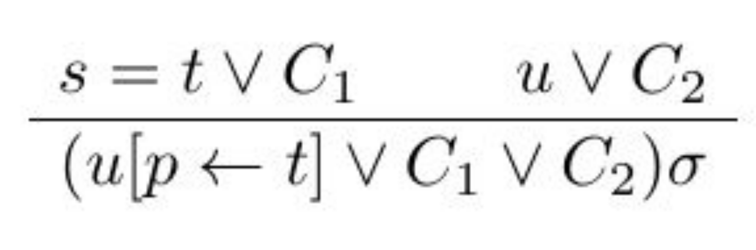
$$\frac{C_1 \lor L_1 \lor L_2}{(C_1 \lor L_2)\sigma}$$

- *Superposition* and *Paramodulation* are fancy terms for combining resolution with *equality reasoning*, the capacity to perform *term rewrites*.
  - For example, if “in_prison(masked_man)” (*u*) and “masked_man = socrates” (*s*=*t*), then we can apply the equality to derive “in_prison(socrates)”.
  - The general form is below, where pσ = sσ (trivially in this case):

$$\frac{s = t \qquad u}{(u[p \leftarrow t])\sigma}$$

Resolution and Superposition Calculi

- *Superposition* and *Paramodulation* are fancy terms for combining resolution with *equality reasoning*, the capacity to perform *term rewrites*.
- For general clauses, such as s = t ⋁ S, we can think of this as a *“conditional equality”*.
  - If S doesn’t hold, then s = t: ¬S → s = t.
  - The general *paramodulation* rule, where σ = mgu(p, s), is:



$$\frac{s = t \lor C_1 \qquad u \lor C_2}{(u[p \leftarrow t] \lor C_1 \lor C_2)\sigma}$$

Resolution and Superposition Calculi

- *Superposition* and *Paramodulation* are fancy terms for combining resolution with *equality reasoning*, the capacity to perform *term rewrites*.
- For general clauses, such as s = t ∨ S, we can think of this as a *“conditional equality”*.
  - If S doesn’t hold, then s = t: ¬S → s = t.
  - The general *paramodulation* rule, where σ = mgu(p, s), is:

$$\frac{s = t \vee C_1 \qquad u \vee C_2}{(u[p \leftarrow t] \vee C_1 \vee C_2)\sigma}$$

- *Superposition* is a term for paramodulation when its use is done in the context of a reduction ordering.
- A *reduction ordering* ensures that term rewriting is only done in one direction, usually toward *smaller* terms. This helps to guarantee *completeness* of the proof search.

Saturation-based ATPs

- The *modus operandi* of saturation-based theorem provers is to take some Axioms (theory), a *negated conjecture*, and to look for a *proof by contradiction*.
- The idea is to keep generating inferences via the superposition calculus until *either* the empty clause (false) is derived or until the search space is saturated, that is, all viable inferences have been performed (which may never happen).
- It’s important to have *reduction rules* that prune or simplify the clauses:
  - E.g., the ATP can delete tautologies and subsumed clauses.
- If the search space is saturated, then there is some model in which the conjecture is false and the axioms are true.

E’s Given Clause Loop

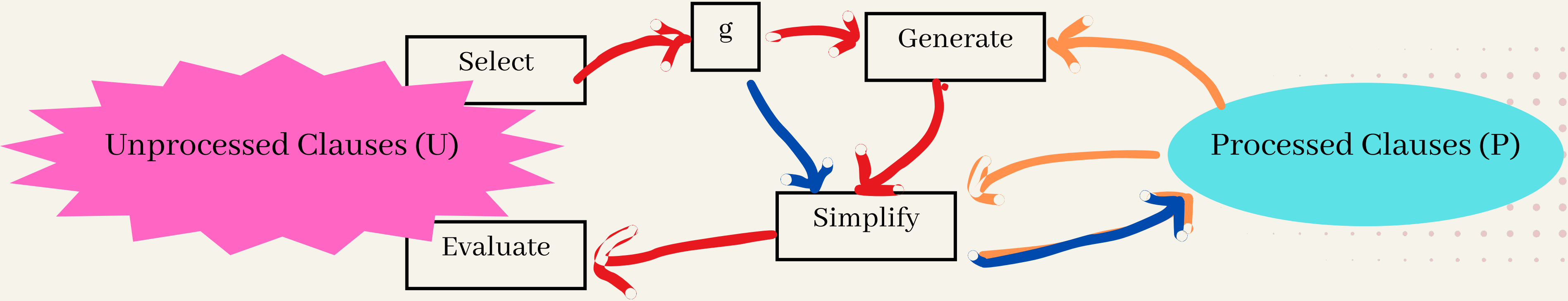
0. Insert the axioms and negated conjecture to the unprocessed clause set *U*.

1. **Select** a given clause *g* from *U* to process and add to the processed set *P*.

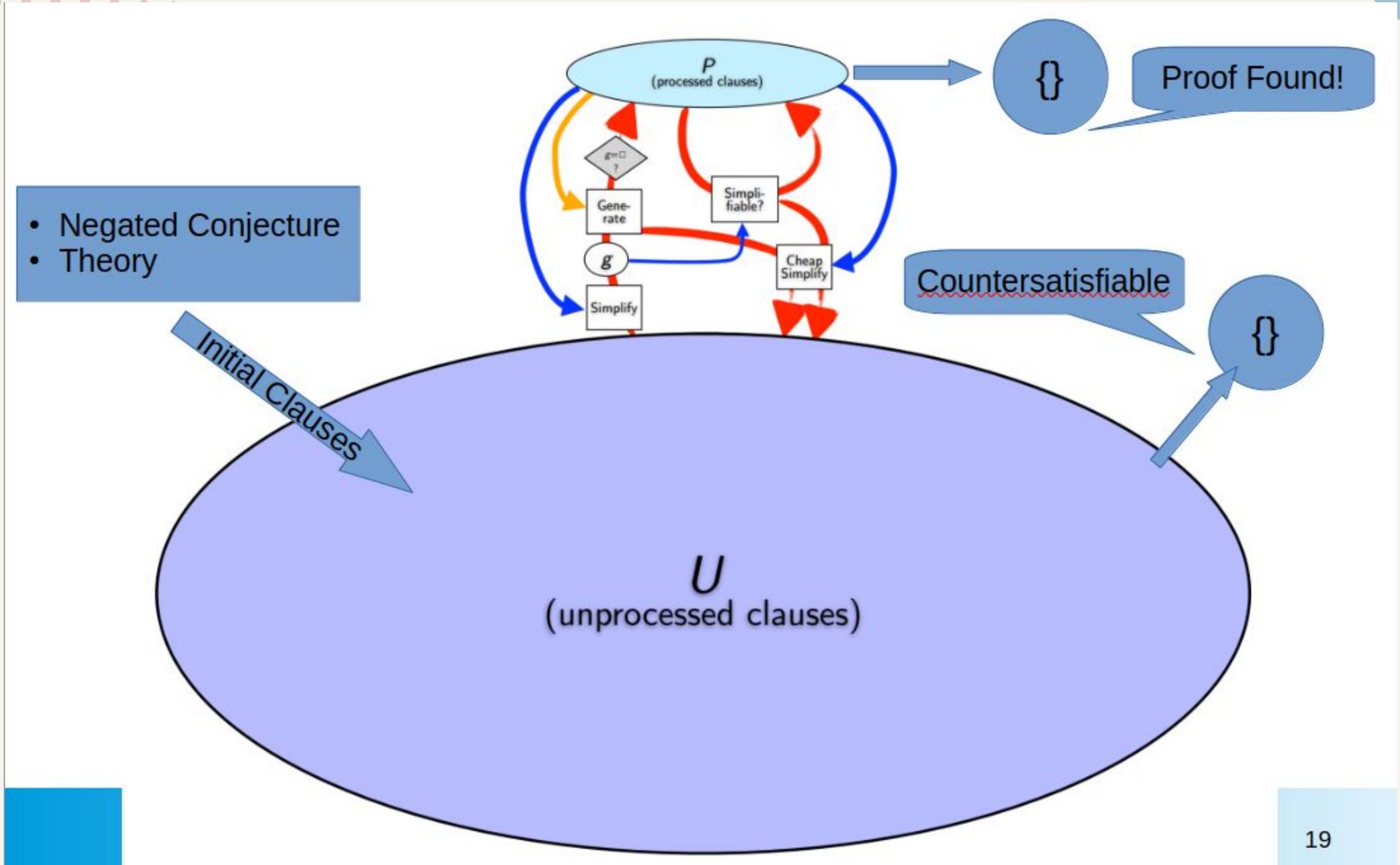
2. **Generate** clauses by performing all inferences between *g* and clauses in *P*.

3. **Simplify** the clauses, remove redundancies, and check for the empty clause.

4. **Evaluate** the simplified generated clauses and add them to *U*.



E’s Given Clause Loop



- Negated Conjecture
- Theory

Initial Clauses

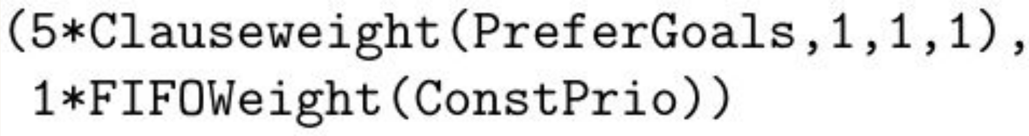
Proof Found!

Countersatisfiable

E Strategies

- Given clauses are selected via *Clause Evaluation Functions* (CEFs).
  - CEFs consist of *priority* and parameterized *weight* functions.
  - Clauses are inserted into a priority queue based on (priority, weight) pairs.
- A strategy is a combination of CEFs following a weighted round-robbin scheme.

- For example,
  - **PreferGoals** selects all goal (negated conjecture) clauses first.
  - **FIFOWeight** assigns increasing weights for a first-in, first-out strategy.
  - **Clauseweight** counts the symbols in a clause to prefer lighter clauses.
  - These make up a classic strategy:



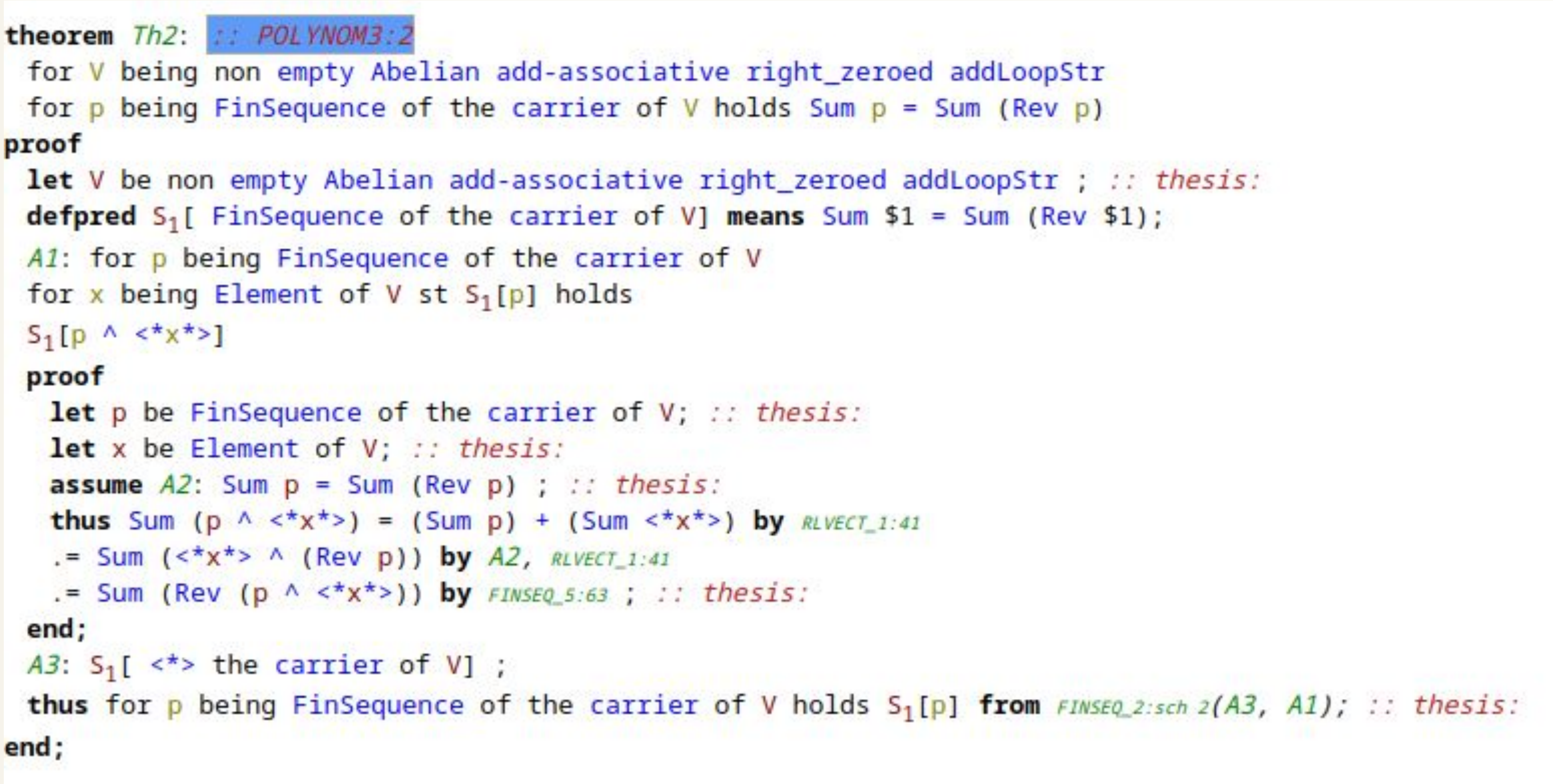
```
(5*Clauseweight(PreferGoals,1,1,1),
 1*FIFOWeight(ConstPrio))
```

Datasets: Mizar Mathematical Library

- **The Mizar Mathematical Library (MML) is one of the largest repositories of formal mathematics.**
- **The Mizar language uses classical first-order logic and the MML is built on top of Tarski-Grothendieck set theory.**
- **The MML contains over 1000 articles on diverse mathematical topics.**
- **Our benchmark consists of 57,880 problems exported into first-order logic by the MPTP system.**
- **Most of the research I've done done is on this benchmark.**
- **You can find some *interesting proofs* on github: https://github.com/ai4reason/ATP_Proofs**

Datasets: Mizar Mathematical Library

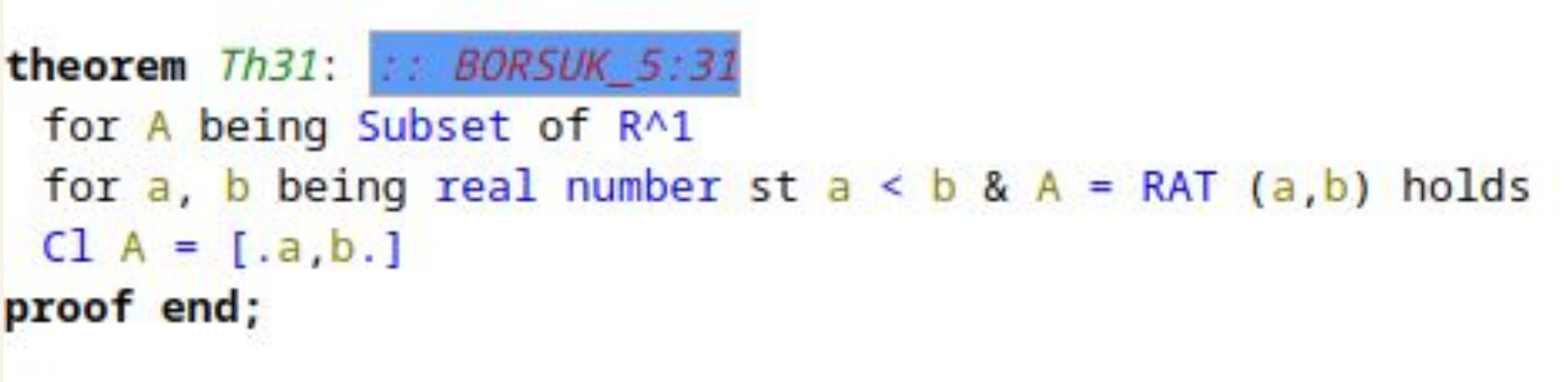
- For a [...] finite sequence *p*, Sum p = Sum (Reverse p).
- Proven with the aid of a GNN (Graph Neural Network) in 5 seconds.



```
theorem Th2: :: POLYNOM3:2
 for V being non empty Abelian add-associative right_zeroed addLoopStr
 for p being FinSequence of the carrier of V holds Sum p = Sum (Rev p)
proof
 let V be non empty Abelian add-associative right_zeroed addLoopStr ; :: thesis:
 defpred S1[ FinSequence of the carrier of V] means Sum $1 = Sum (Rev $1);
 A1: for p being FinSequence of the carrier of V
 for x being Element of V st S1[p] holds
 S1[p ^ <*x*>]
 proof
  let p be FinSequence of the carrier of V; :: thesis:
  let x be Element of V; :: thesis:
  assume A2: Sum p = Sum (Rev p) ; :: thesis:
  thus Sum (p ^ <*x*>) = (Sum p) + (Sum <*x*>) by RLVECT_1:41
  .= Sum (<*x*> ^ (Rev p)) by A2, RLVECT_1:41
  .= Sum (Rev (p ^ <*x*>)) by FINSEQ_5:63 ; :: thesis:
 end;
 A3: S1[ <*> the carrier of V] ;
 thus for p being FinSequence of the carrier of V holds S1[p] from FINSEQ_2:sch 2(A3, A1); :: thesis:
end;
```

Datasets: Mizar Mathematical Library

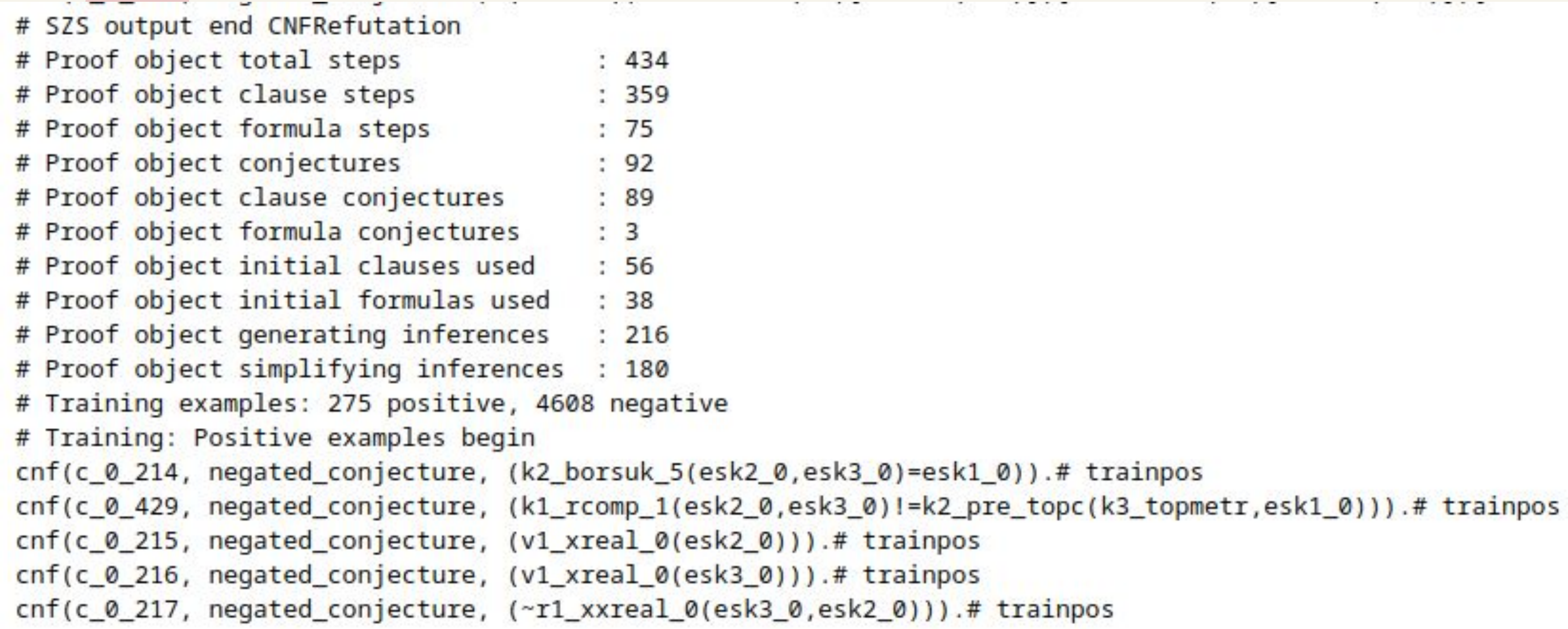
- The closure of rationals on (a,b) is [a,b].



```
theorem Th31: :: BORSUK_5:31
  for A being Subset of R^1
  for a, b being real number st a < b & A = RAT (a,b) holds
  Cl A = [.a,b.]
proof end;
```

Datasets: Mizar Mathematical Library

- The closure of rationals on (a,b) is [a,b].
- Proven with 3 AI models: the GNN and decision trees in two roles in 234 second.



```
# SZS output end CNFRefutation
# Proof object total steps             : 434
# Proof object clause steps            : 359
# Proof object formula steps           : 75
# Proof object conjectures             : 92
# Proof object clause conjectures      : 89
# Proof object formula conjectures     : 3
# Proof object initial clauses used    : 56
# Proof object initial formulas used   : 38
# Proof object generating inferences   : 216
# Proof object simplifying inferences  : 180
# Training examples: 275 positive, 4608 negative
# Training: Positive examples begin
cnf(c_0_214, negated_conjecture, (k2_borsuk_5(esk2_0,esk3_0)=esk1_0)).# trainpos
cnf(c_0_429, negated_conjecture, (k1_rcomp_1(esk2_0,esk3_0)!=k2_pre_topc(k3_topmetr,esk1_0))).# trainpos
cnf(c_0_215, negated_conjecture, (v1_xreal_0(esk2_0))).# trainpos
cnf(c_0_216, negated_conjecture, (v1_xreal_0(esk3_0))).# trainpos
cnf(c_0_217, negated_conjecture, (~r1_xxreal_0(esk3_0,esk2_0))).# trainpos
```

Datasets: Isabelle Sledgehammer Problems

- Isabelle/HOL is one of the largest proof assistants that uses a simple type theory-based higher-order logic (HOL).
- The Sledgehammer system exports goals to the TPTP language for ATPs and reconstructs the proofs in Isabelle.
- Our dataset consisted of:
  - 1902 theory files
  - 276,363 problems
  - 179 Isabelle/Mirabelle sessions
    - 80 from the AFP (Archive of Formal Proofs)
    - 75 from Isabelle 2021-1
    - 24 from IsaFoR (Isabelle Formalization of Rewriting)
- See Yutaka’s talk for more info.

Learning in Theorem Proving

1. High-level:
   a. Premise selection of lemmas from a mathematical library.
   b. Select/evolve strategies (for specific problems/domains).
   c. Find *hints* for a problem to guide ATPs.
   d. Develop proof sketches (possibly from related theories).
   e. Feedback loops learning over low & mid-level tasks.
   f. Autoformalization: translate natural language (latex) to logic.
2. Mid-level:
   a. Invent intermediate lemmas, concepts, or models for a problem.
   b. Guide the application of tactics in an Interactive Theorem Proving system (ITP)
   c. Learn new tactics (like program-synthesis).
3. Low-level:
   a. Guide every inference step (-- what I'll mainly talk about –).

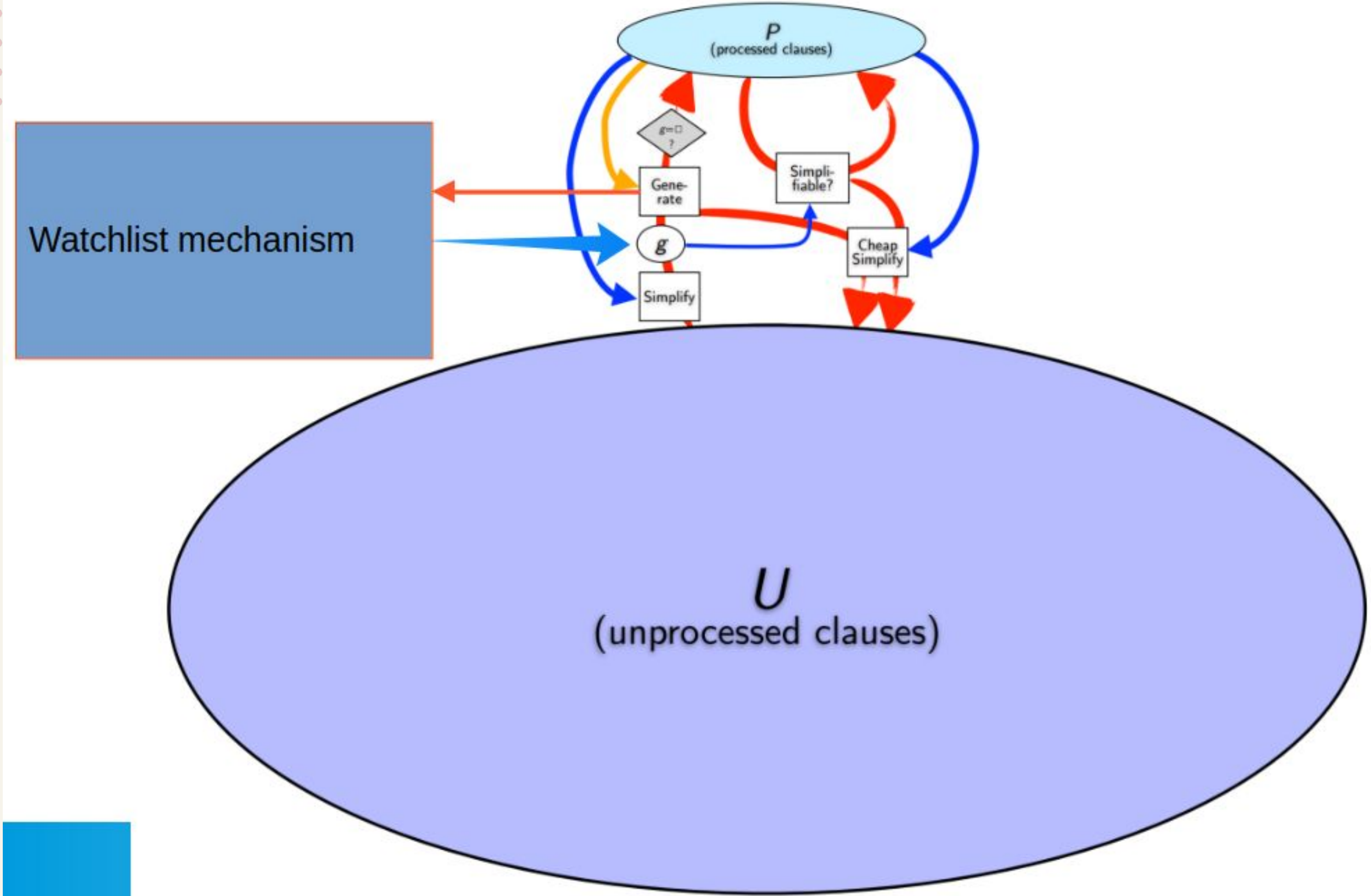


Image thanks to Stephan Schulz’s presentation on E

The Watchlist Technique

- A symbolic guidance method invented by Veroff.
- Each proof by contradiction has the same goal: the empty clause {}.
- Maintain a list of lemmas, often from proofs of related theorems.
  - This is called the watchlist or hint list.
- A generated clause matches a hint if it subsumes the hint.
- Prioritize clauses that match hints for selection.

ENIGMA Features

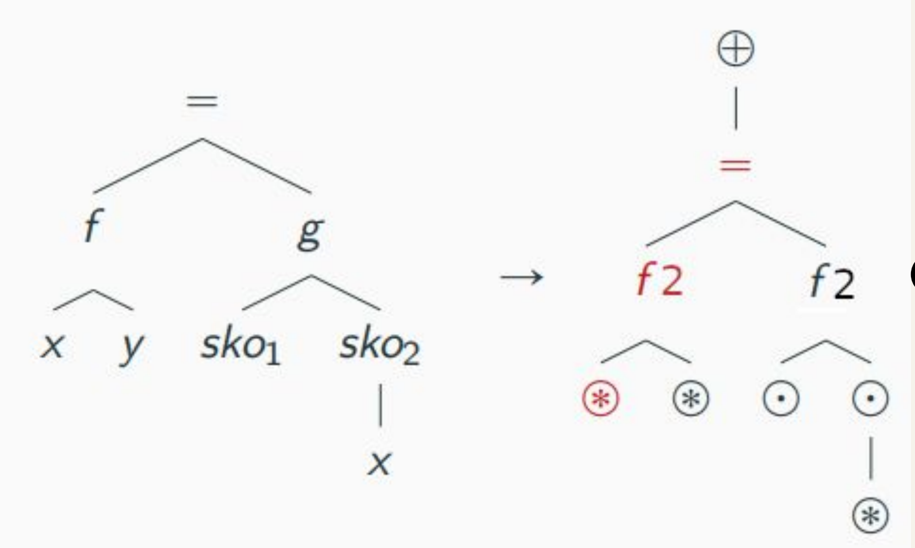
- To work with many AI techniques, we need vectorial features.
    - Even for the k-NN based hint list selection.

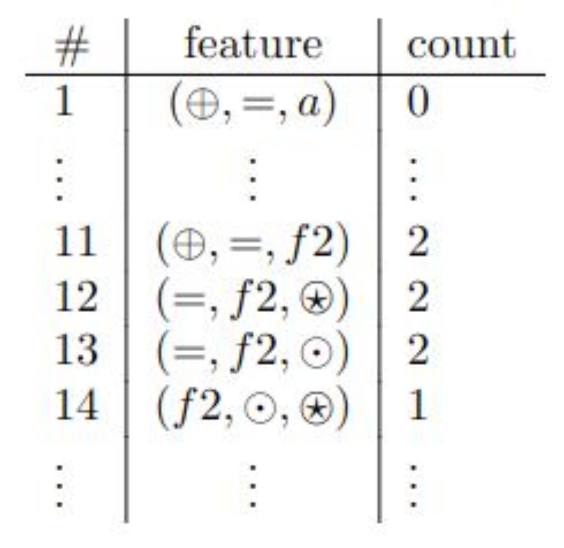
- *ENIGMA features* aim to capture syntactic structure as a multi-set of features.
    - Represent clauses as feature vectors derived from term trees.
    - Vertical features are top-down term walks of length 3.

ENIGMA Features

- Vertical feature example: the clause vector consists of feature counts.

$$f(x, y) = g(\mathrm{sko}_1, \mathrm{sko}_2(x))$$





#	feature	count
1	$(\oplus, =, a)$	0
⋮	⋮	⋮
11	$(\oplus, =, f2)$	2
12	$(=, f2, \circledast)$	2
13	$(=, f2, \odot)$	2
14	$(f2, \odot, \circledast)$	1
⋮	⋮	⋮

ENIGMA Features

- Aim to capture syntactic structure as a multi-set of features.
    - Represent clauses as feature vectors derived from term trees.

- Some optional features:
    - Horizontal features include the term and top-level argument symbols.
    - Anonymization of function and predicate symbol names by only using their arity.
    - Hash features to reduce dimensionality.
    - Features from the conjecture clauses and parent clauses.

ProofWatch

Core ideas:

- Extend Veroff’s watchlist technique.
  - Maintain a list of proofs on the watchlist.
  - Track the completion ratio for each proof, based on how much it’s matched.
  - Dynamically assign higher priority to clauses matching more complete proofs.

ProofWatch

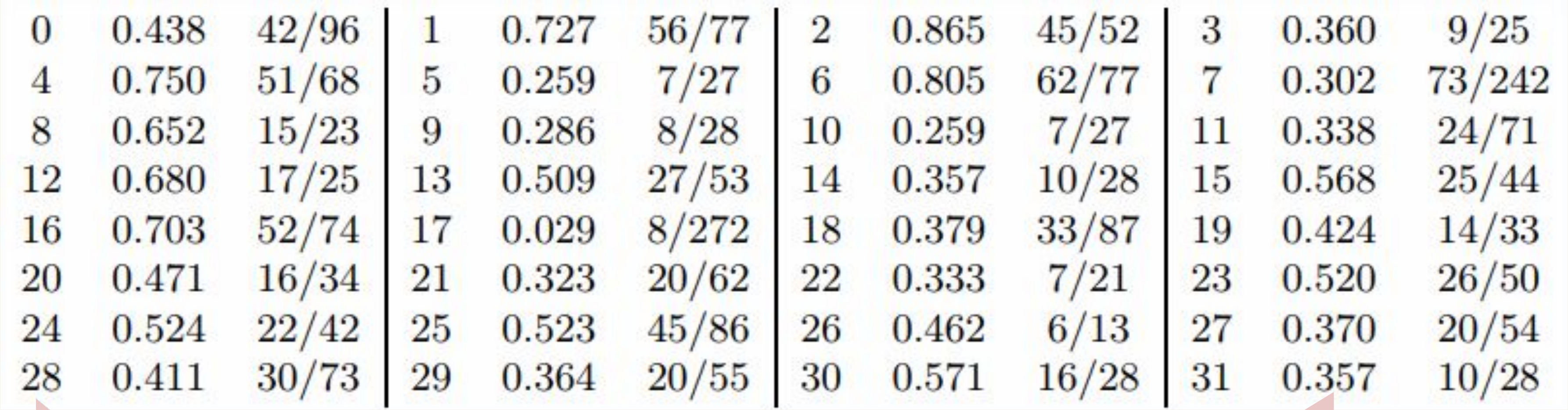
Core ideas:

- Extend Veroff’s watchlist technique.
  - Maintain a list of proofs on the watchlist.
  - Track the completion ratio for each proof, based on how much it’s matched.
  - Dynamically assign higher priority to clauses matching more complete proofs.
- This proof vector aims to capture the semantic space of the proof search.
- Use k-nearest neighbors to recommend proofs to add to the watchlist.
- Resembles episodic memory.
- Achieved 26% improvement on MML (the Mizar Mathematical Library)

Proof Vector

A snapshot of the successfully useful proof-vector for YELLOW 5:36

(De Morgan’s laws) with 32 k-NN recommended proofs of related problems:



0	0.438	42/96	1	0.727	56/77	2	0.865	45/52	3	0.360	9/25
4	0.750	51/68	5	0.259	7/27	6	0.805	62/77	7	0.302	73/242
8	0.652	15/23	9	0.286	8/28	10	0.259	7/27	11	0.338	24/71
12	0.680	17/25	13	0.509	27/53	14	0.357	10/28	15	0.568	25/44
16	0.703	52/74	17	0.029	8/272	18	0.379	33/87	19	0.424	14/33
20	0.471	16/34	21	0.323	20/62	22	0.333	7/21	23	0.520	26/50
24	0.524	22/42	25	0.523	45/86	26	0.462	6/13	27	0.370	20/54
28	0.411	30/73	29	0.364	20/55	30	0.571	16/28	31	0.357	10/28

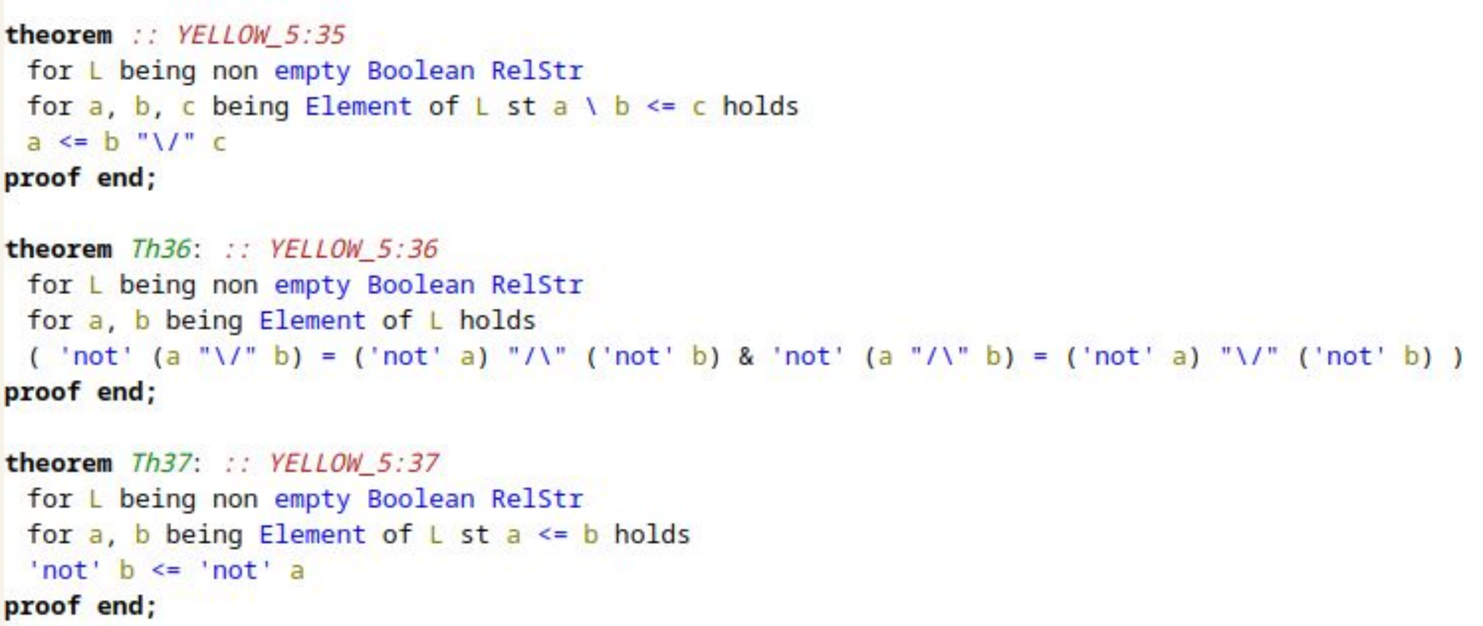
Proof Number

Completion Ratio

Proof Vector

A snapshot of the successfully useful proof-vector for YELLOW 5:36

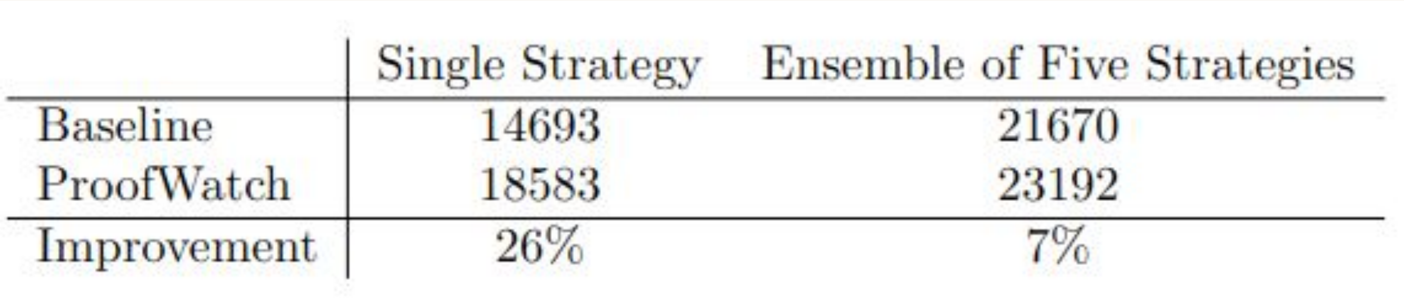
(De Morgan’s laws) with 32 k-NN recommended proofs of related problems:



```
theorem :: YELLOW_5:35
 for L being non empty Boolean RelStr
 for a, b, c being Element of L st a \ b <= c holds
 a <= b "\/" c
proof end;

theorem Th36: :: YELLOW_5:36
 for L being non empty Boolean RelStr
 for a, b being Element of L holds
 ( 'not' (a "\/" b) = ('not' a) "/\" ('not' b) & 'not' (a "/\" b) = ('not' a) "\/" ('not' b) )
proof end;

theorem Th37: :: YELLOW_5:37
 for L being non empty Boolean RelStr
 for a, b being Element of L st a <= b holds
 'not' b <= 'not' a
proof end;
```

ProofWatch Results

- The results are on a benchmark of 57897 Mizar problems.

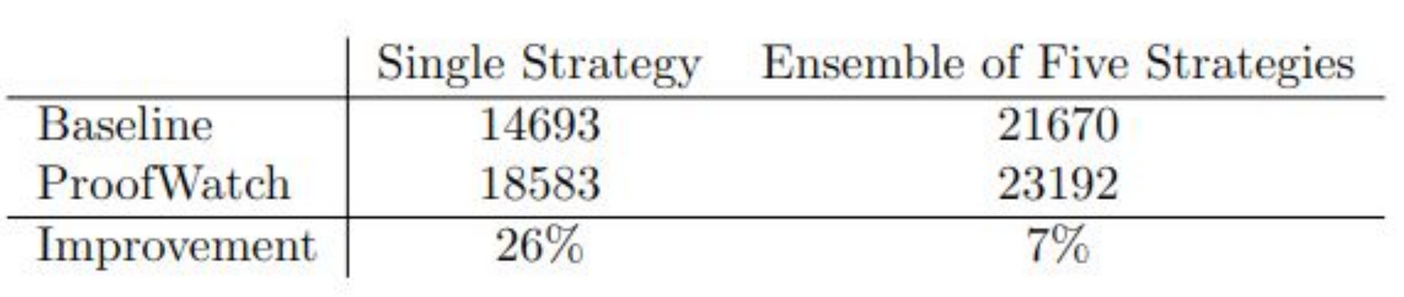


	Single Strategy	Ensemble of Five Strategies
Baseline	14693	21670
ProofWatch	18583	23192
Improvement	26%	7%

- The best results were obtained with the dynamic watchlist feature with 16 k-NN recommended proofs.

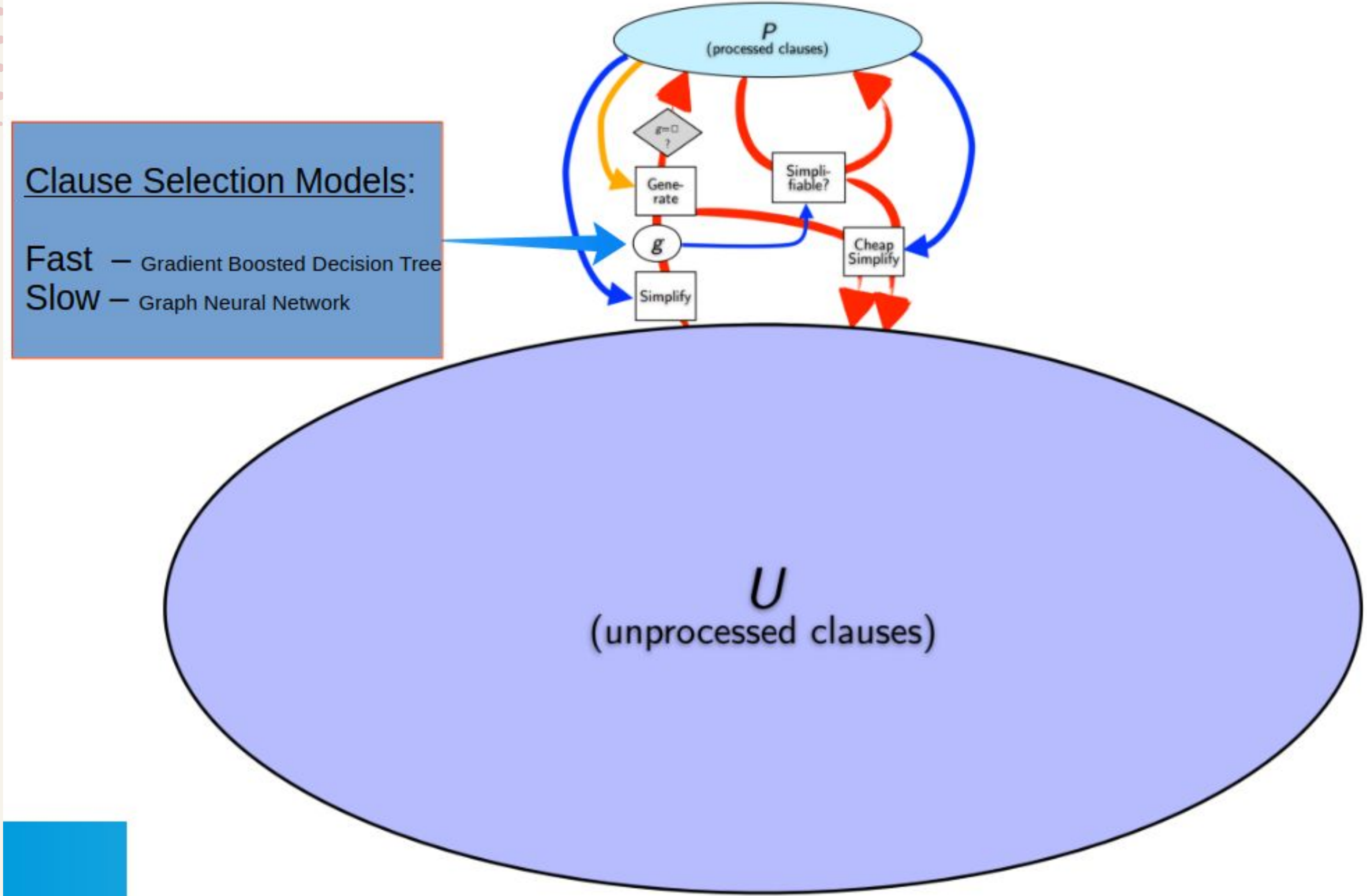
ProofWatch Results

- The results are on a benchmark of 57897 Mizar problems.



	Single Strategy	Ensemble of Five Strategies
Baseline	14693	21670
ProofWatch	18583	23192
Improvement	26%	7%

🌟 The ATP field still has low-hanging fruit if the topic excites you! 🌟



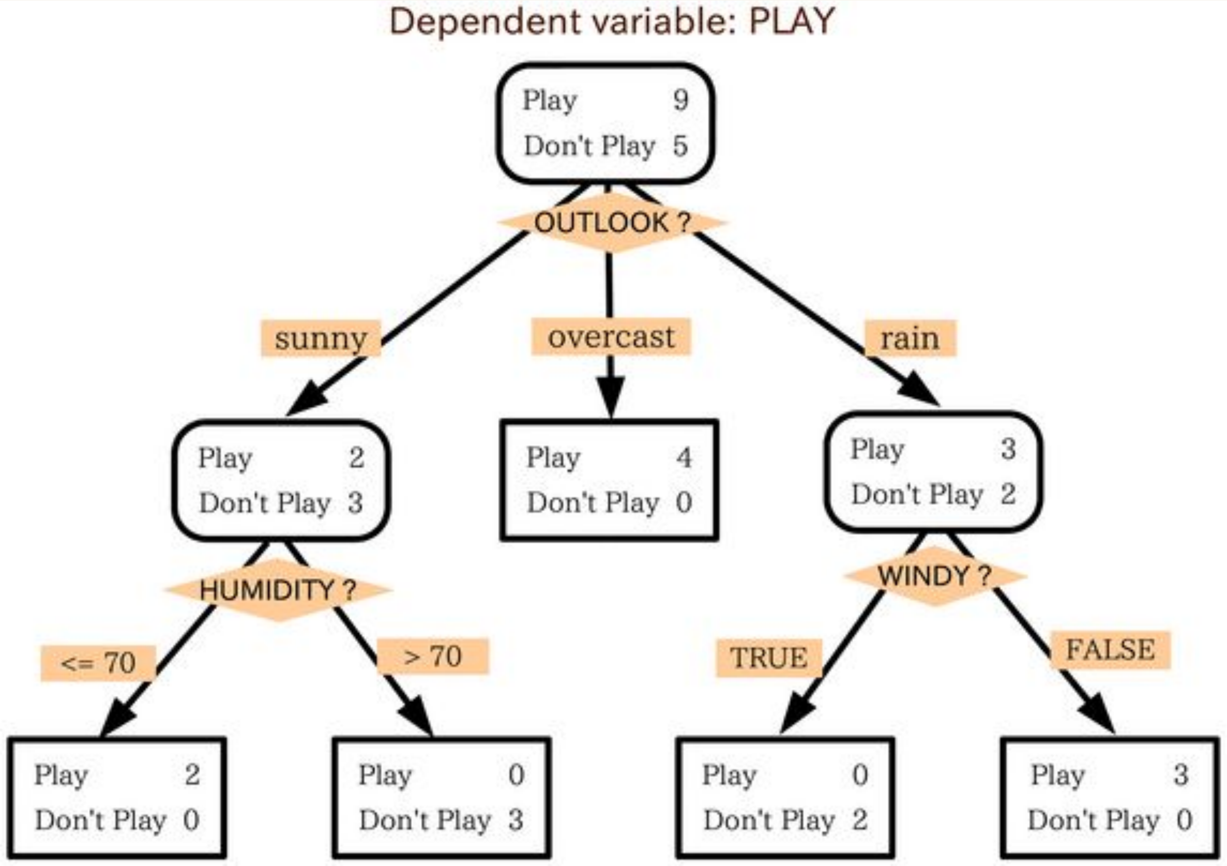
Clause Selection Models:

Fast – Gradient Boosted Decision Tree
Slow – Graph Neural Network

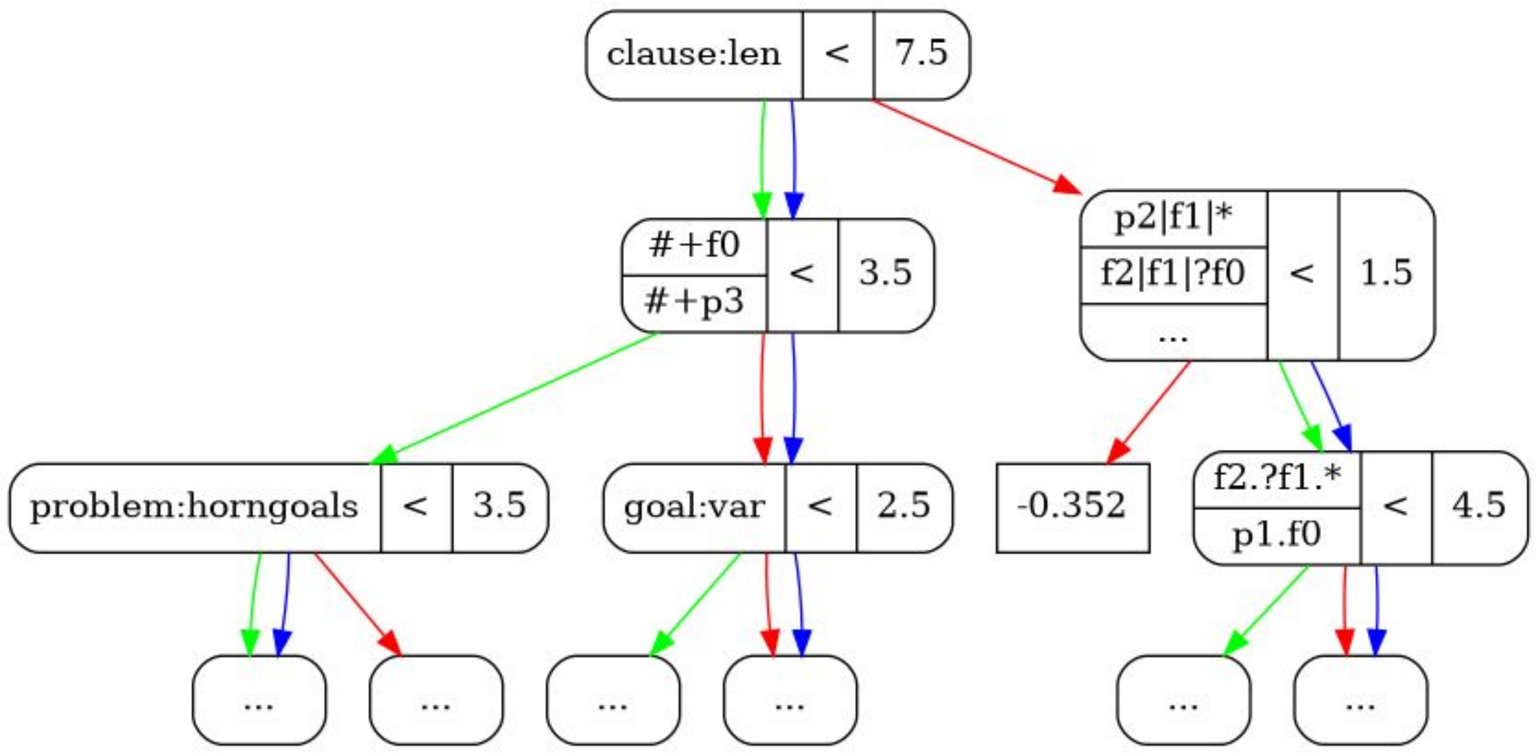
Gradient Boosted Decision Trees

- A statistical learning method typically used for binary classification.
- In contrast to the symbolic methods, it needs features (as vectors).
- Decision nodes are added to trees in order to maximally discriminate between *positive* and *negative* examples in the training data set.
- Each new tree in the ensemble is trained based on the performance (‘gradient’) of the previous trees.
- Libraries used:
    - XGBoost
    - LightGBM

Gradient Boosted Decision Tree



Gradient Boosted Decision Tree



Gradient Boosted Decision Trees

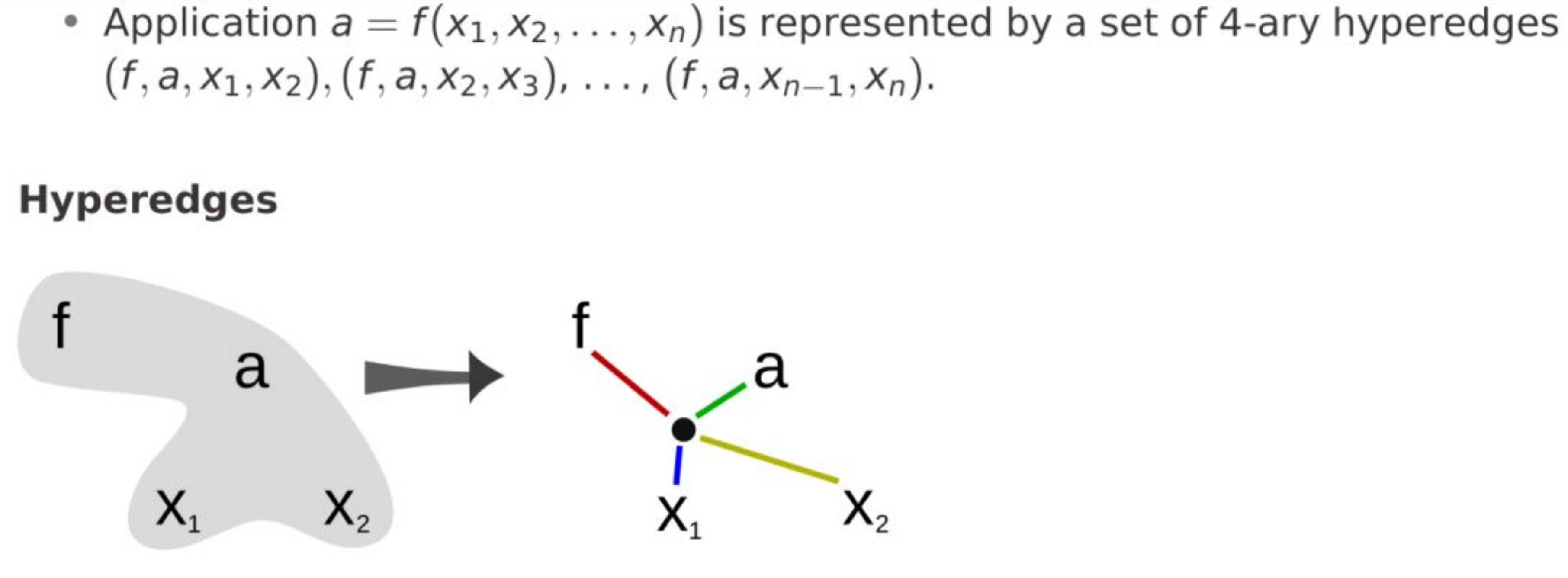
- A statistical learning method typically used for binary classification.
- In contrast to the symbolic methods, it needs features (as vectors).
- Decision nodes are added to trees in order to maximally discriminate between *positive* and *negative* examples in the training data set.
- Each new tree in the ensemble is trained based on the performance (‘gradient’) of the previous trees.
- Libraries used:
    - XGBoost: grows full trees level-by-level.
    - LightGBM: grows trees one leaf at a time (so they may be imbalance).
        - We’ve found LightGBM to deliver faster training with good performance.

Graph Neural Network

- Directed hypergraph for a set of clauses
- Anonymized symbol names:
  - The network only captures the mathematical structure.
- Nodes: clauses, functions and predicate symbols, unique (sub)terms, and literals.
- Hyperedges:
  1. Clauses and literals
  2. Functions and predicates with subterms

Graph Neural Network

- Argument ordering is (partially) preserved:



- Application $a = f(x_1, x_2, \ldots, x_n)$ is represented by a set of 4-ary hyperedges $(f, a, x_1, x_2), (f, a, x_2, x_3), \ldots, (f, a, x_{n-1}, x_n)$.

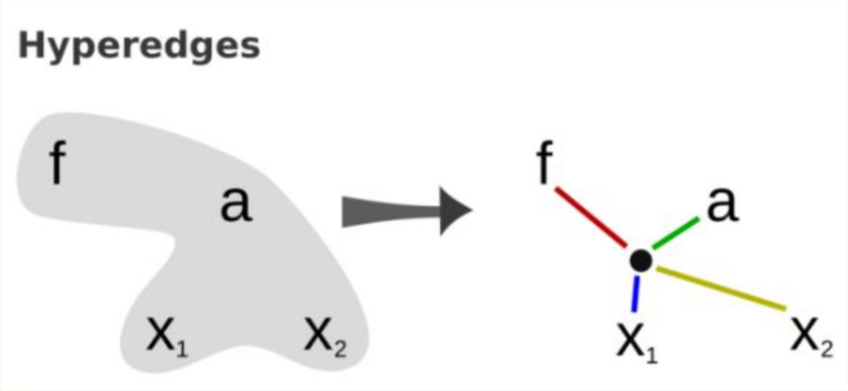
Hyperedges

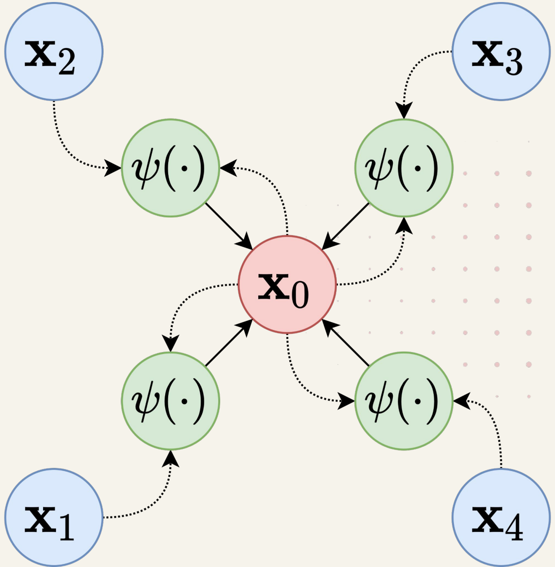
Graph Neural Network

- Message passing rounds follow formula structure:

  → Node ‘embeddings’ are updated via edges from one layer to the next.

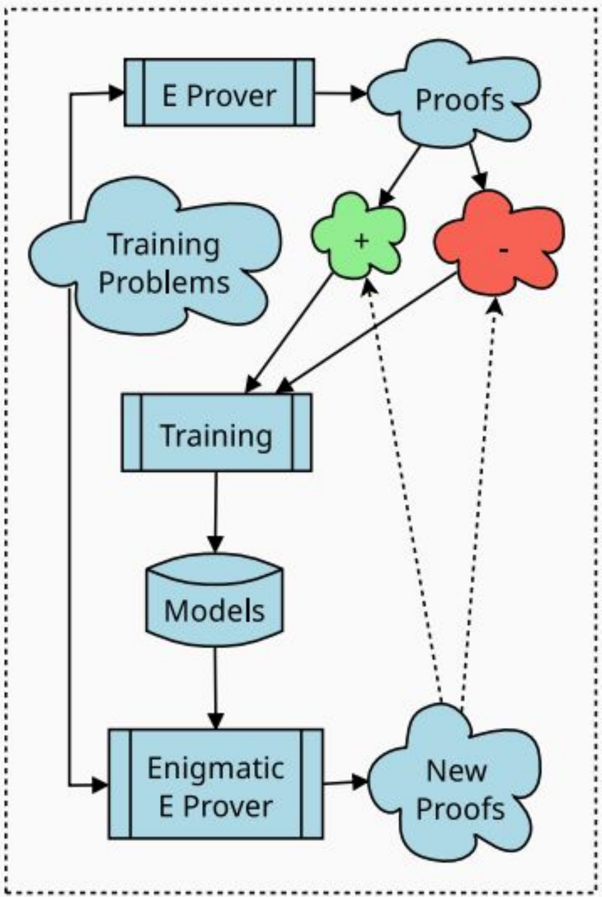
  → Gives us clause embeddings and a prediction layer.





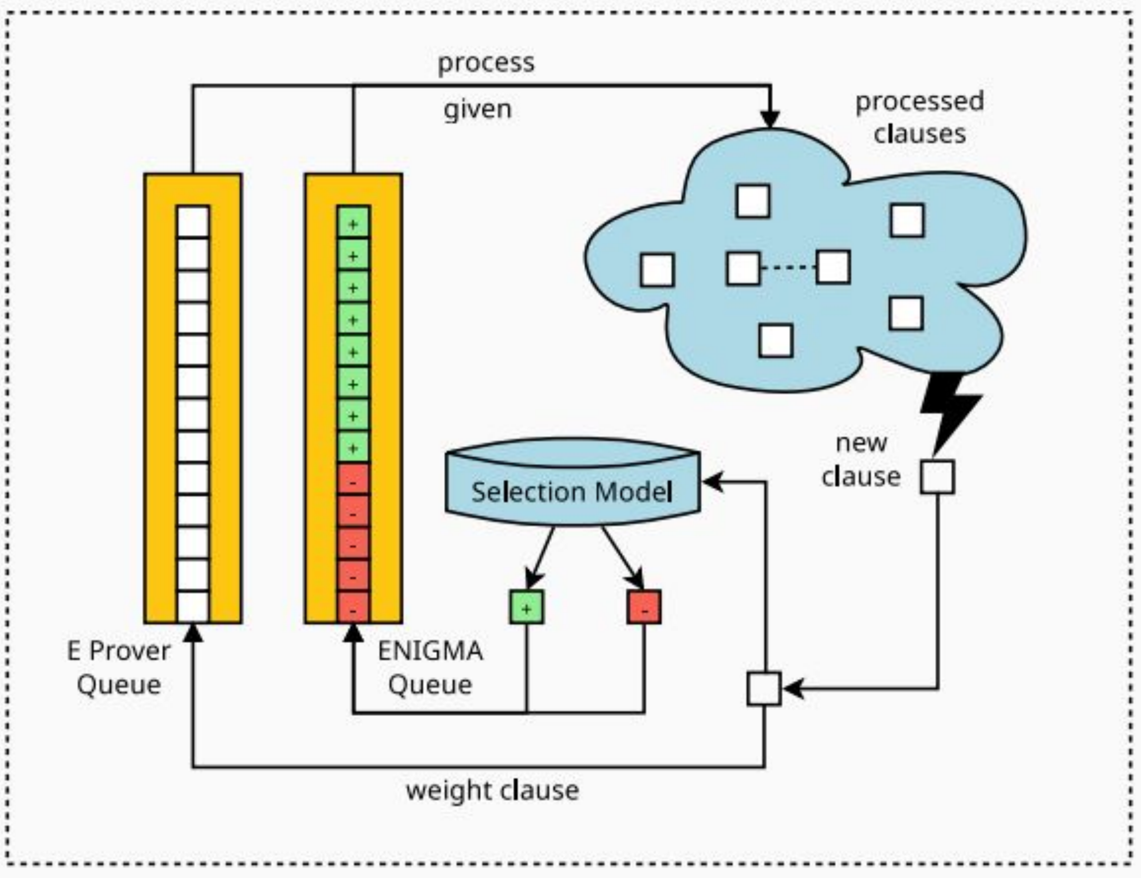
ENIGMA: Training Loop

1. Run E over a training dataset.
2. Harvest training samples from the proofs:
   - positive = proof clauses
   - negative = other (processed) clauses
3. Train a classification model.
4. Plug the model into E and go to 1.



ENIGMA: Given Clause Loop

- The ENIGMA model is usually used in parallel with standard heuristics (in a cooperative way).

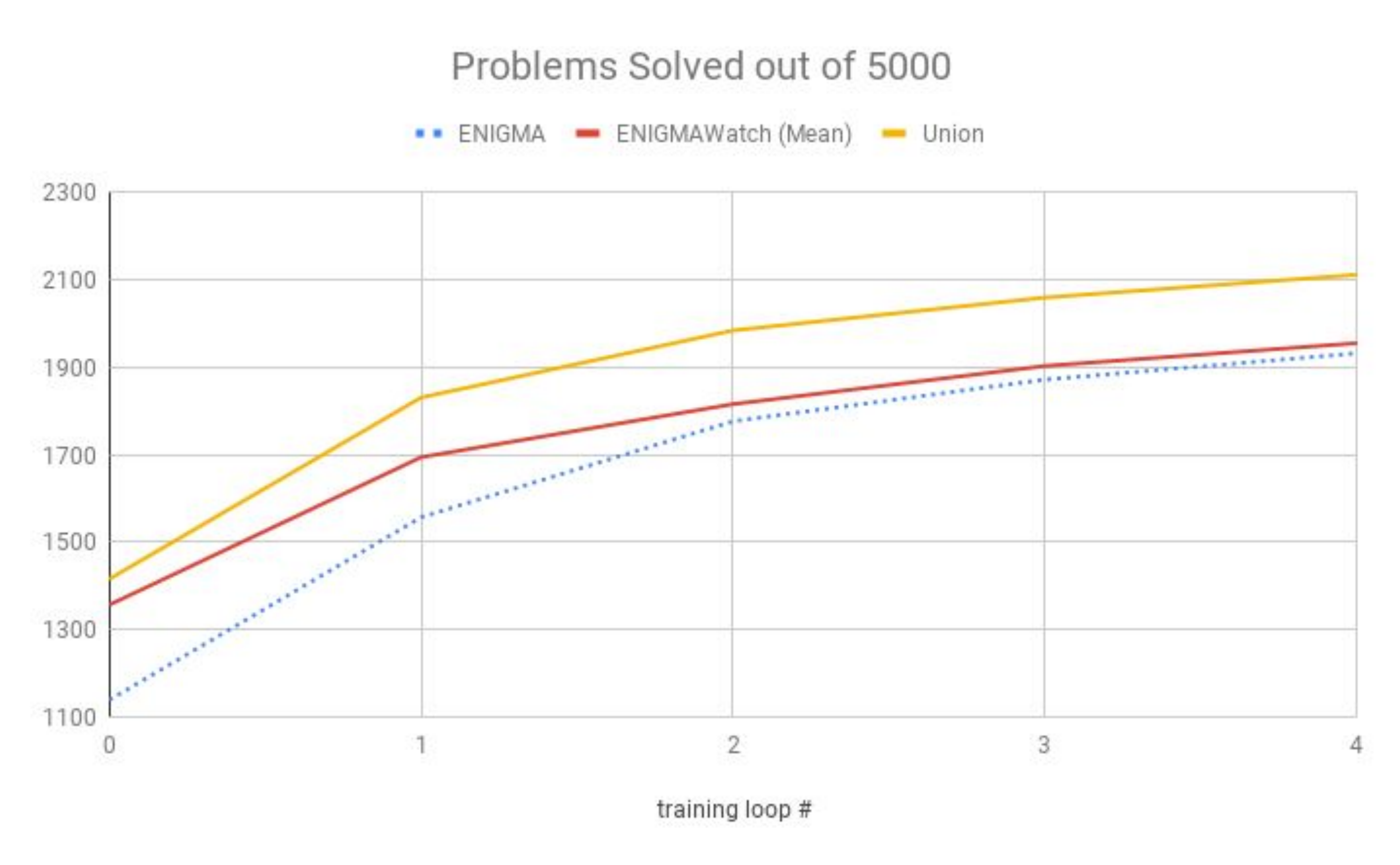


ENIGMA + ENIGMAWatch

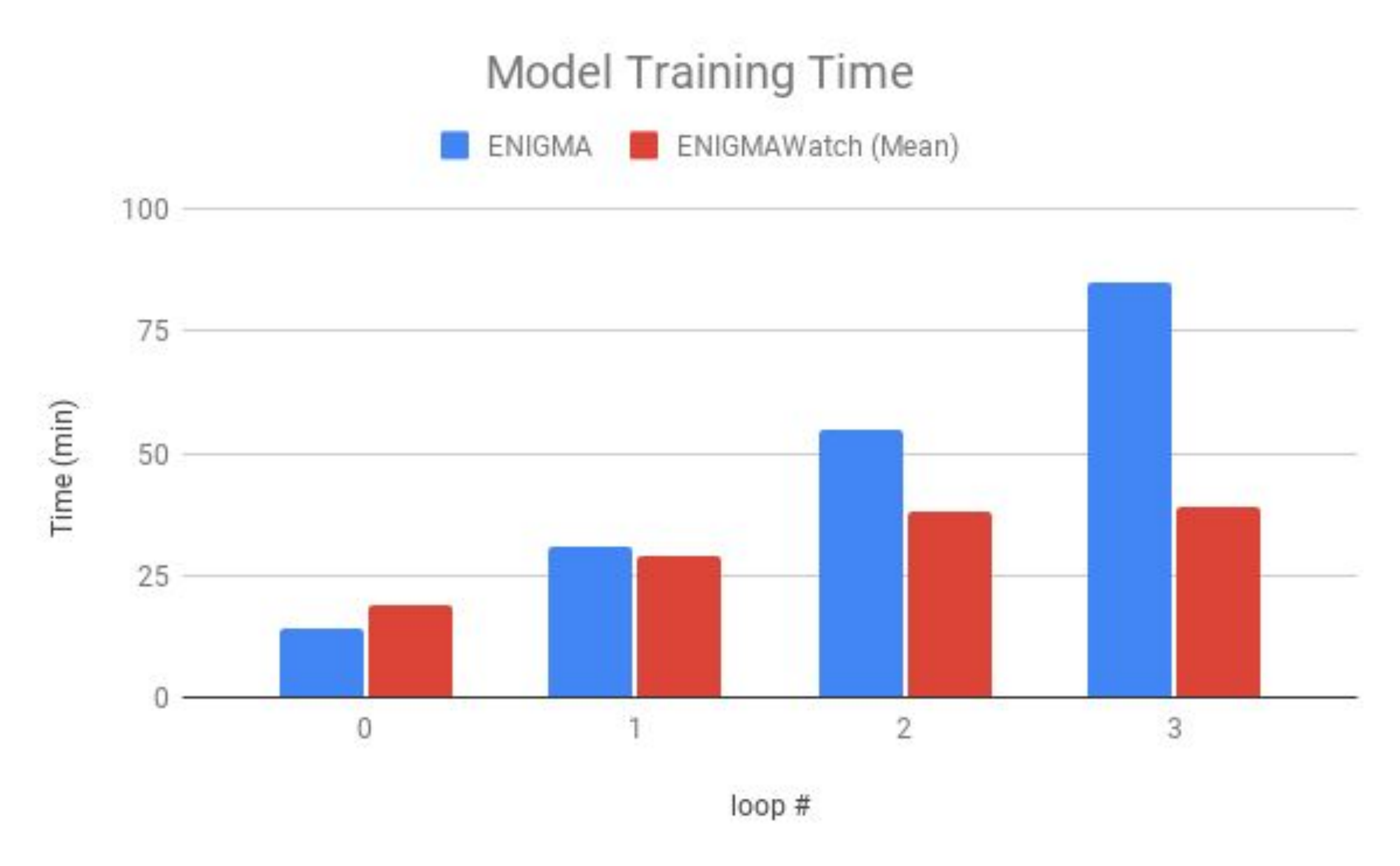
Core ideas:

- ENIGMA:
    - Train predictive models based on completed proof searches.
    - Use gradient-boosted decision tree libraries.
        - Or the GNN.
    - Combines with E’s native search strategies.
- ENIGMAWatch:
    - Use the ProofWatch proof vector as additional input to ENIGMA.
    - While using the watchlist guidance.
    - Explored various methods of creating proof vectors.

ENIGMAWatch Results



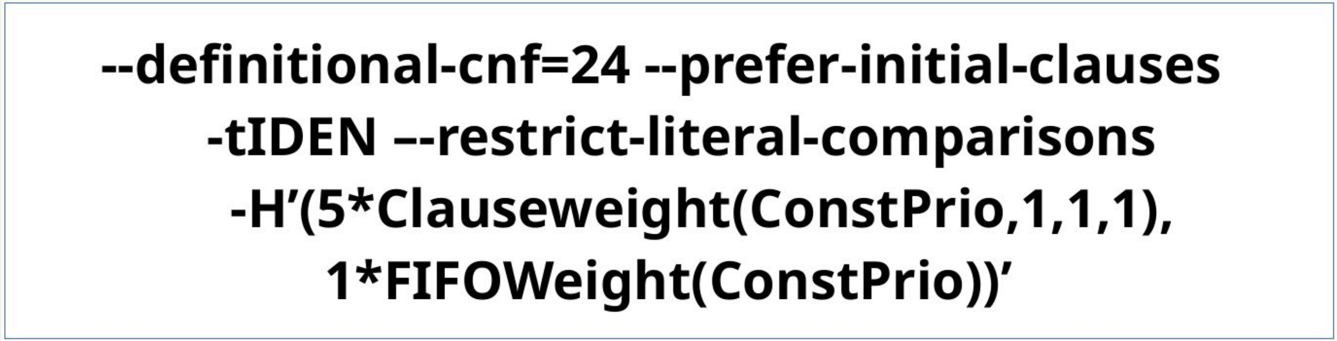
ENIGMAWatch Results



Make E Smart Again

Core questions:

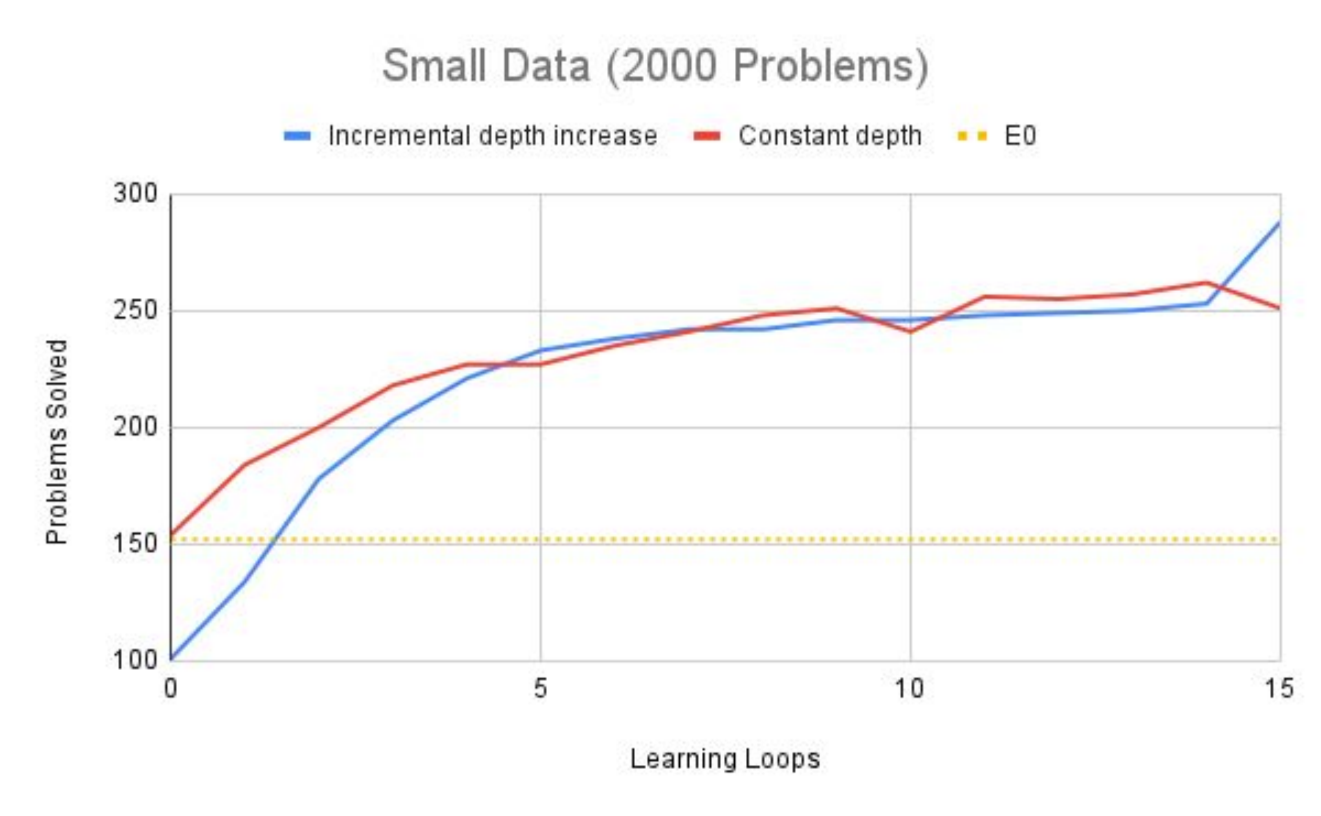
- Can ENIGMA learn to guide E without symmetry breaking methods or good search strategies?
  - Nearly disable term ordering (and thus some term rewriting rules).
  - Restrict literal orderings, used for literal selection.
  - Use only



```
--definitional-cnf=24 --prefer-initial-clauses
-tIDEN –-restrict-literal-comparisons
-H’(5*Clauseweight(ConstPrio,1,1,1),
1*FIFOWeight(ConstPrio))’
```

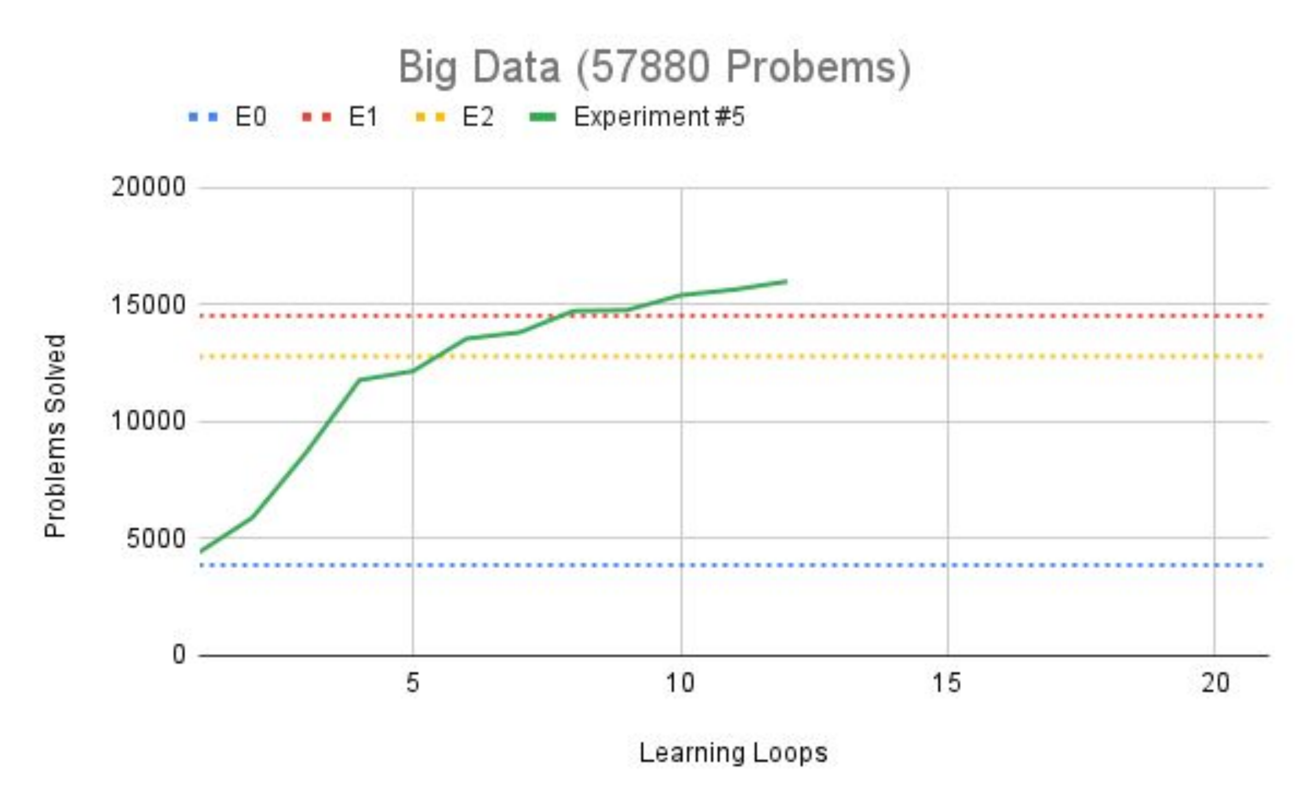
Make E Smart Again - Results

- Incrementally increasing the depth of trees in the ENIGMA models outperformed looping with a constant tree depth.



Make E Smart Again - Results

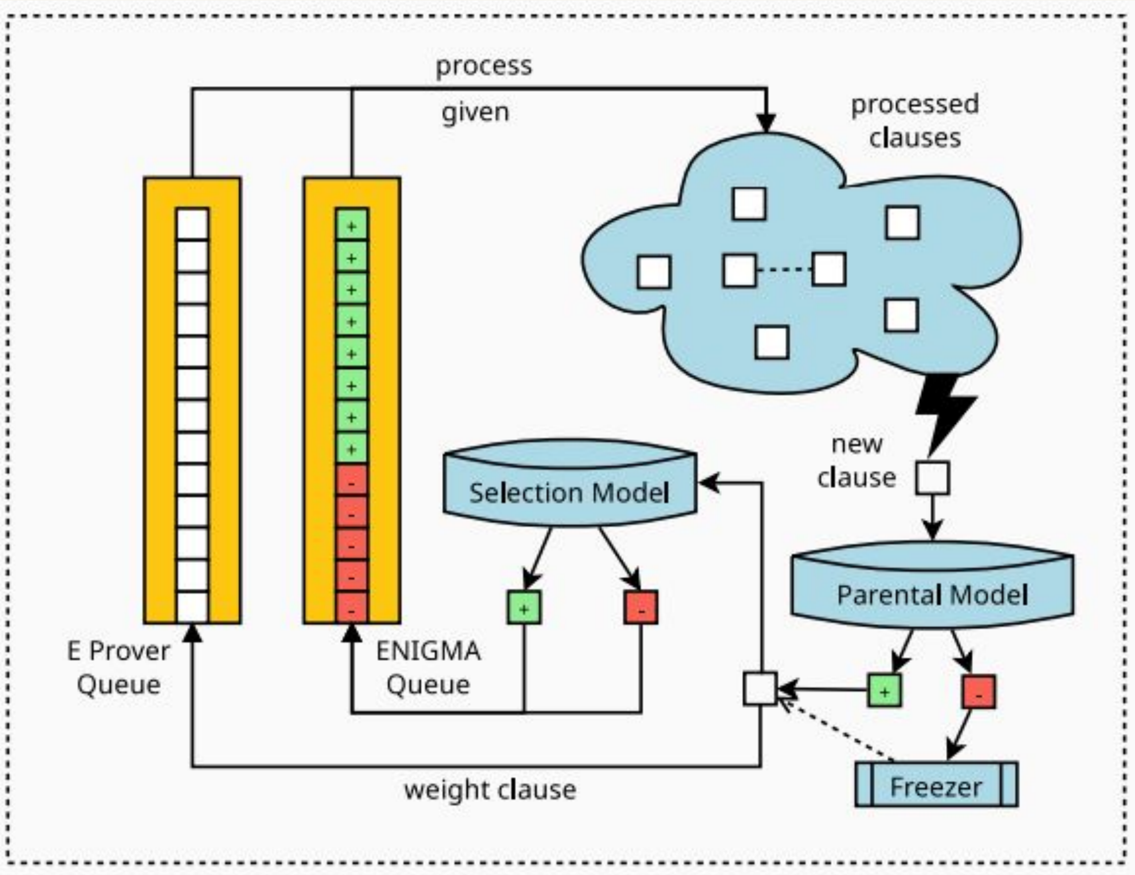
- E0 is the baseline simple strategy.
- E1 and E2 are two strong E strategies that use the KBO term-ordering with good literal selection strategies.
- Experiment #5 surpasses E1 after 11 loops of training and its first loop is boosted with data from an ENIGMA model in coop with E1.



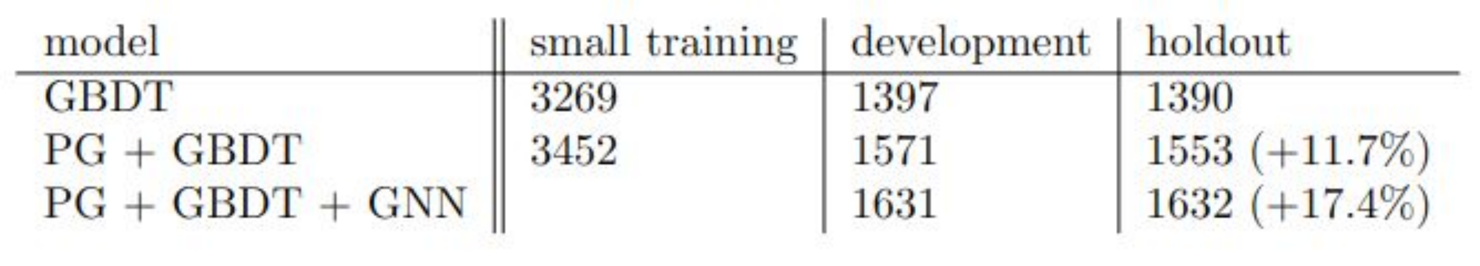
Parental Guidance

Core ideas:

- The features of a clause's parents may help in evaluating a clause.
- Filtering clauses immediately post-generation may save resources.
- A pre-filter before the other strategies may be a good component of a multi-model system.



Parental Guidance Results



model	small training	development	holdout
GBDT	3269	1397	1390
PG + GBDT	3452	1571	1553 (+11.7%)
PG + GBDT + GNN		1631	1632 (+17.4%)

- Results on a small training set of 5792 problems and development and holdout sets of 2896 problems each. Models are run for 30 seconds.
- Parental Guidance (PG) run in combination with the best gradient-boosted decision tree (GBDT) model attains 11.7% higher performance.
- 3-phase ENIGMA adds a GNN for clause selection to PG and GDBT, attaining the best performance.
- E’s auto-schedule proved 1020 problems on holdout, making for a 60% improvement.

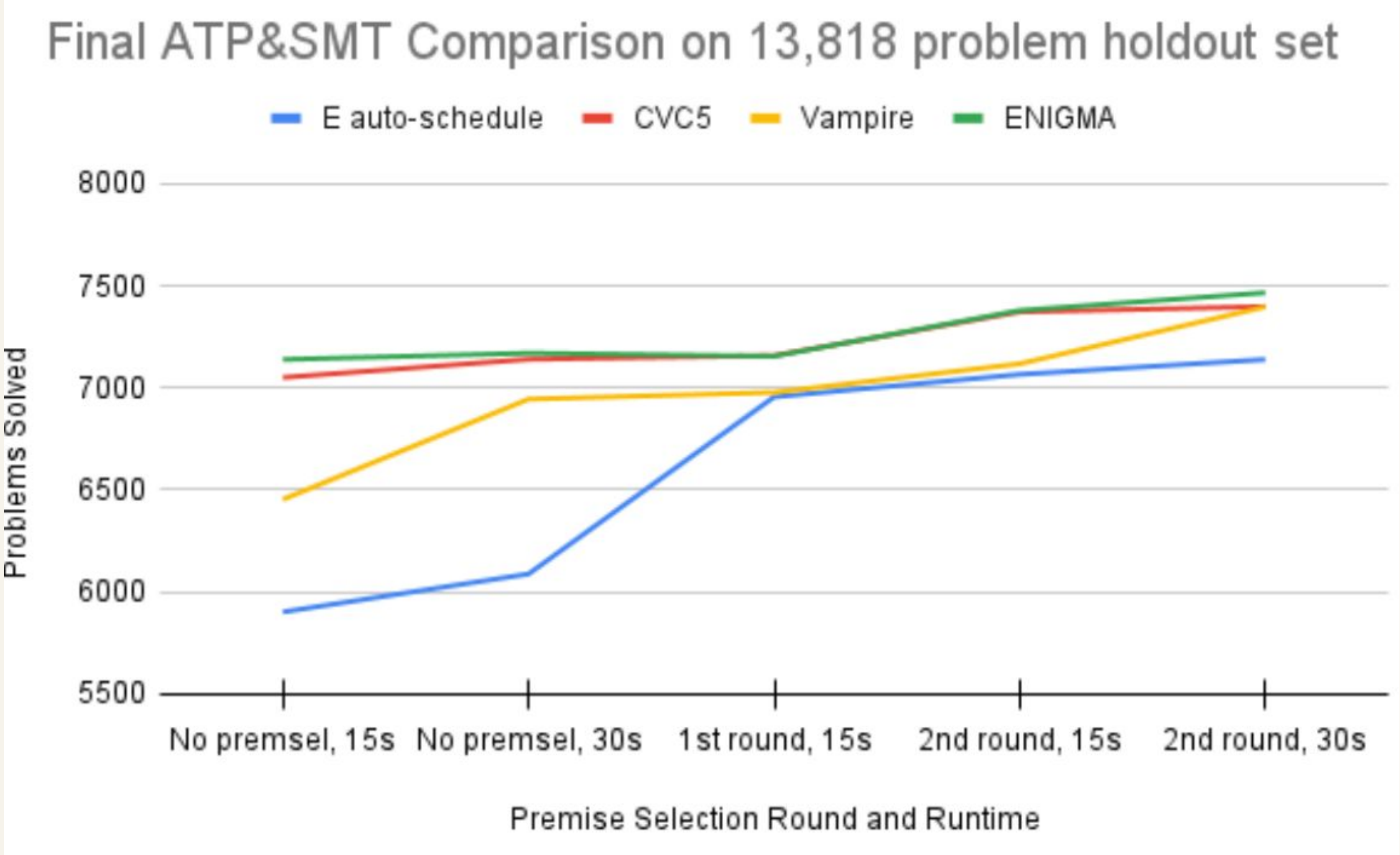
The Isabelle ENIGMA

Core question:

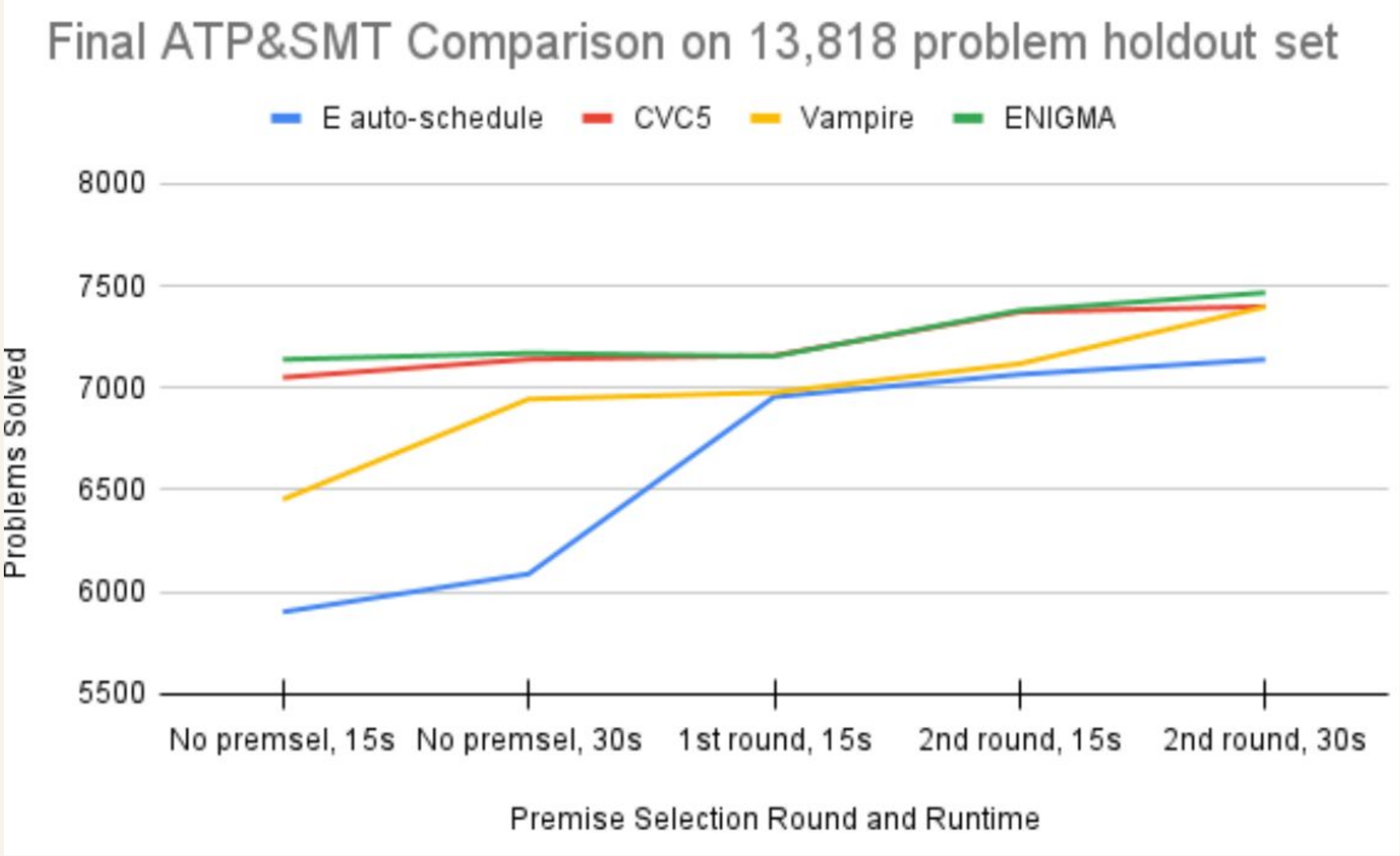
- Does ENIGMA transfer from the Mizar Mathematical Library to other formal math libraries?

- The dataset’s 276,363 problems are split into
  - 248k training problems
  - 13.8k problems for devel and holdout sets
- We used a symbol-anonymous graph neural network for premise selection.

The Isabelle ENIGMA Results



The Isabelle ENIGMA Results



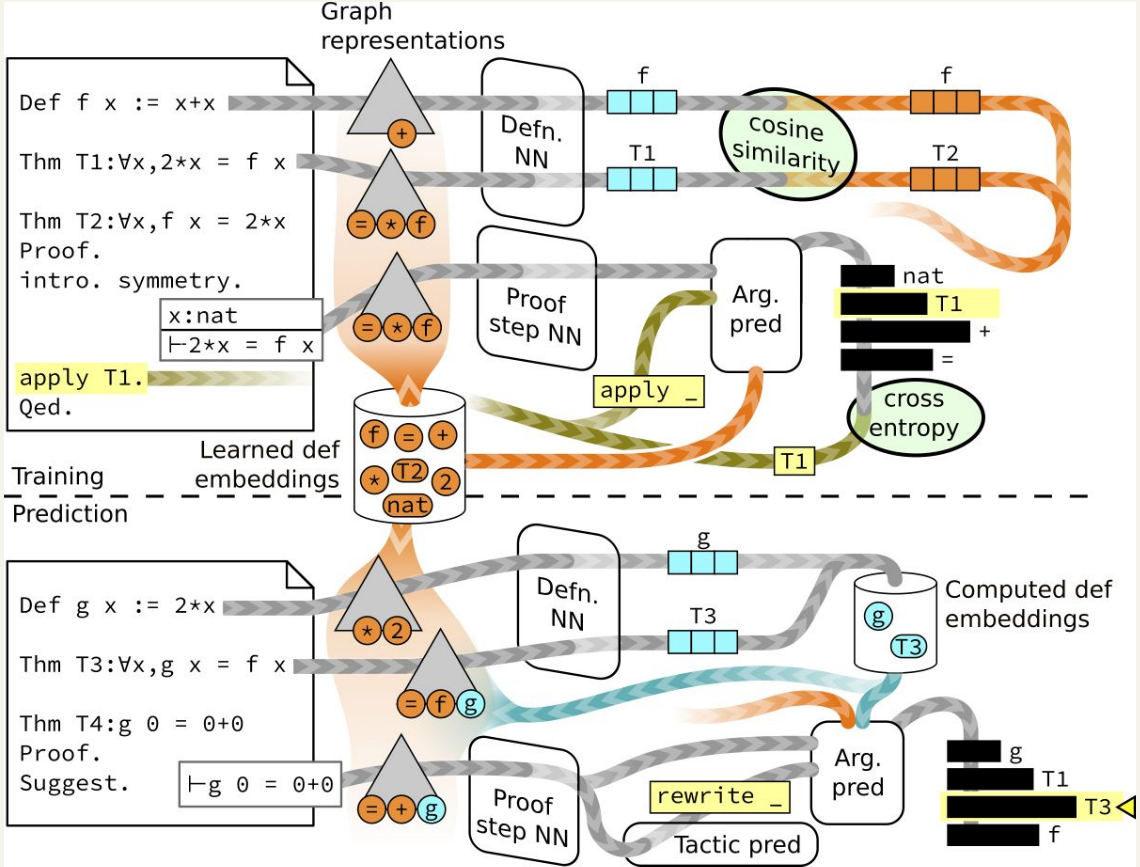
CVC5 ENIGMA

- I hear ENIGMA is being ported to CVC5 now with good success.
- See First Experiments with Neural cvc5 if curious.

The Tactician

- **The Tactician is an interactive tactic learner for Coq.**
- **Coq is an Interactive Theorem Proving (ITP) system that works with the *calculus of constructions*, higher-order type theory.**
  - **It’s implemented in OCaml.**
- ***Tactics* are programmatic commands that change the proof state (toward a proof).**
- **The Tactician learns from previously written tactic scripts in the Coq library or in the current file!**
- **Its best learners are: k-NN and Graph2Tac.**

Graph2Tac

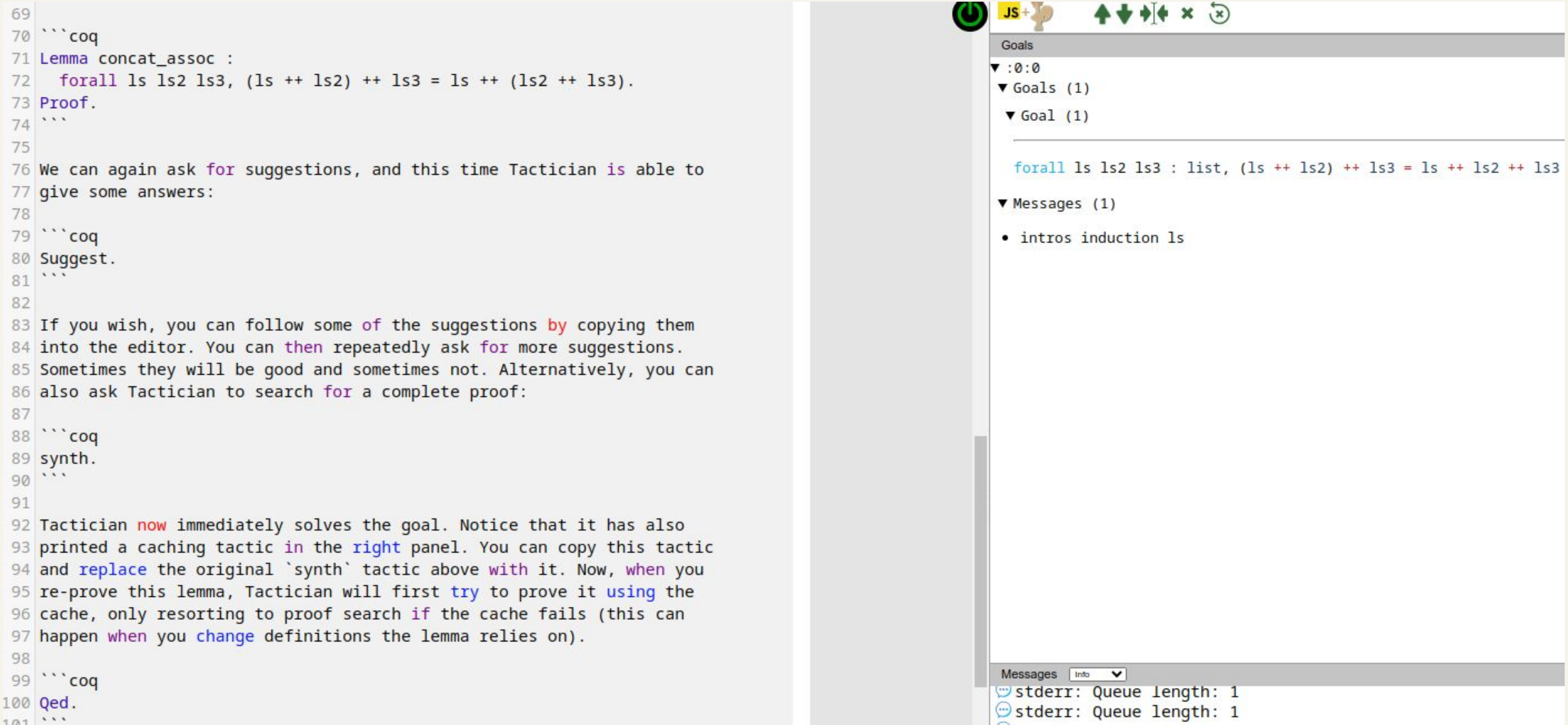


```
Def f x := x+x

Thm T1:∀x,2*x = f x

Thm T2:∀x,f x = 2*x
Proof.
intro. symmetry.
apply T1.
Qed.
```

```
Def g x := 2*x

Thm T3:∀x,g x = f x

Thm T4:g 0 = 0+0
Proof.
Suggest.
```

The Tactician: Demo

- There’s a demo at: https://coq-tactician.github.io/demo.html.



~~~
```coq
Lemma concat_assoc :
  forall ls ls2 ls3, (ls ++ ls2) ++ ls3 = ls ++ (ls2 ++ ls3).
Proof.
```

We can again ask for suggestions, and this time Tactician is able to
give some answers:

```coq
Suggest.
```

If you wish, you can follow some of the suggestions by copying them
into the editor. You can then repeatedly ask for more suggestions.
Sometimes they will be good and sometimes not. Alternatively, you can
also ask Tactician to search for a complete proof:

```coq
synth.
```

Tactician now immediately solves the goal. Notice that it has also
printed a caching tactic in the right panel. You can copy this tactic
and replace the original `synth` tactic above with it. Now, when you
re-prove this lemma, Tactician will first try to prove it using the
cache, only resorting to proof search if the cache fails (this can
happen when you change definitions the lemma relies on).

```coq
Qed.
```
~~~

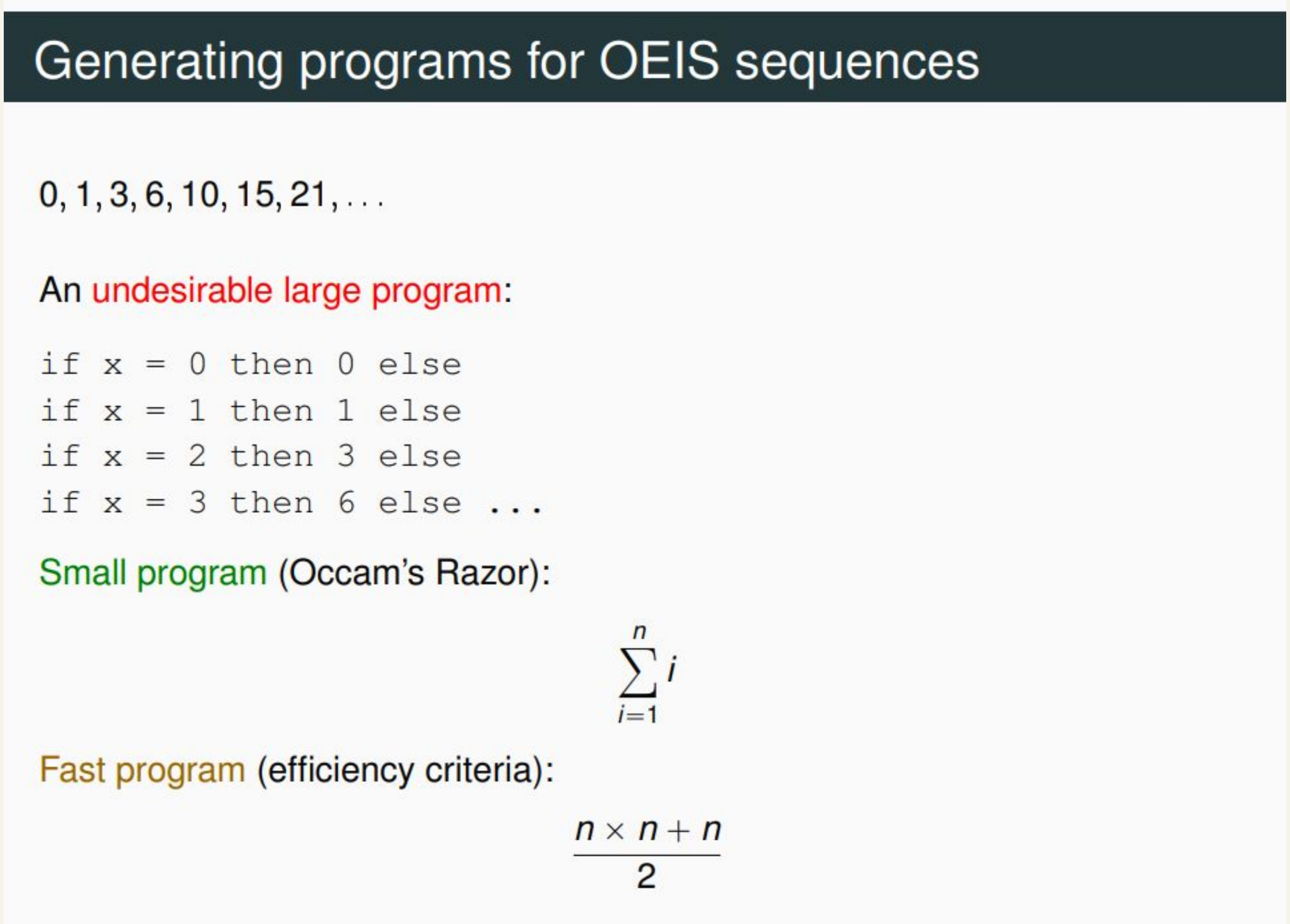
Goals

```
▼ :0:0
 ▼ Goals (1)
  ▼ Goal (1)

   forall ls ls2 ls3 : list, (ls ++ ls2) ++ ls3 = ls ++ ls2 ++ ls3

 ▼ Messages (1)

  • intros induction ls
```

Messages Info

```
stderr: Queue length: 1
stderr: Queue length: 1
```

QSynt: Program Synthesis for Integer Sequences



Generating programs for OEIS sequences

0, 1, 3, 6, 10, 15, 21, . . .

An undesirable large program:

```
if x = 0 then 0 else
if x = 1 then 1 else
if x = 2 then 3 else
if x = 3 then 6 else ...
```

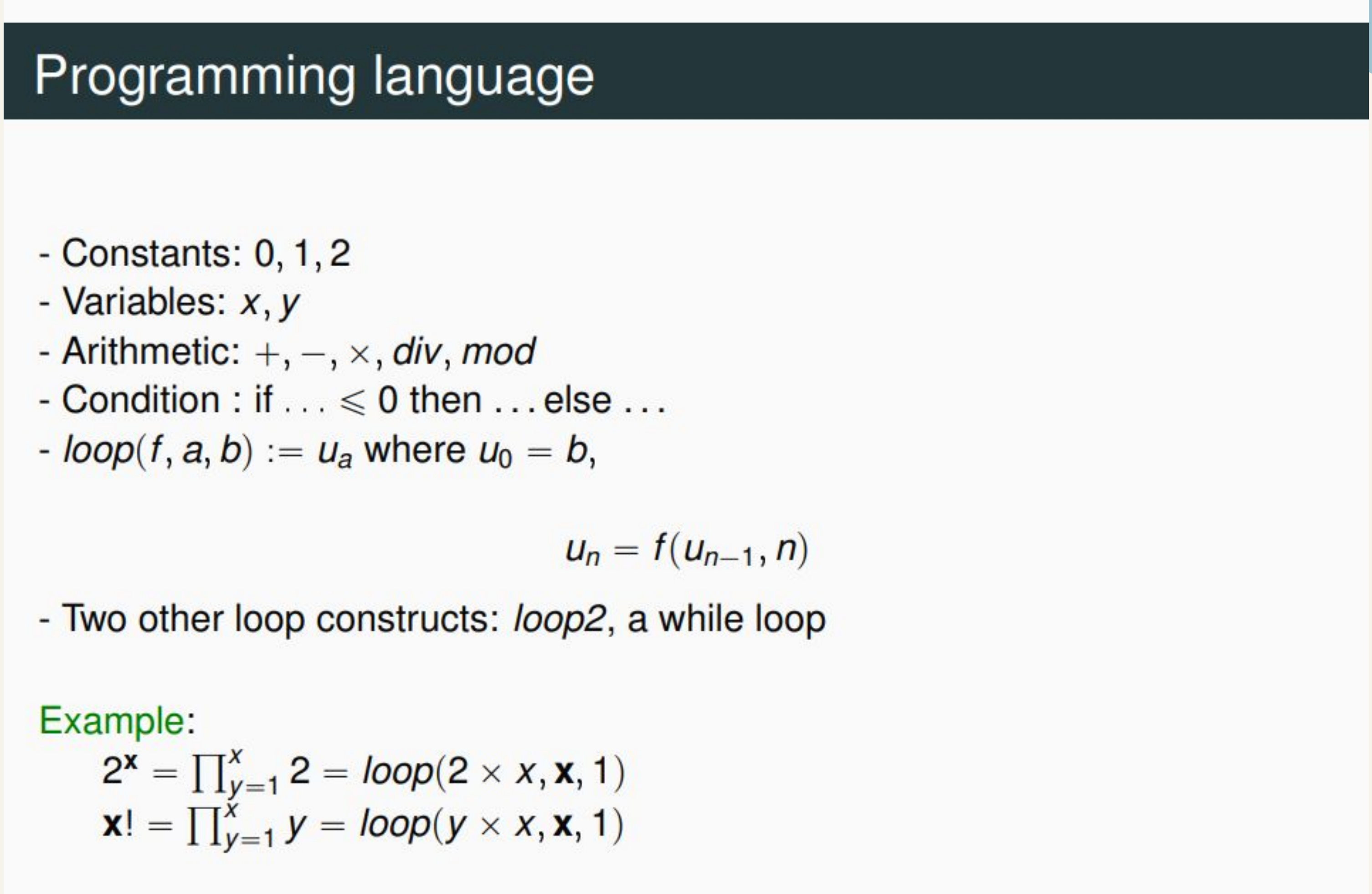
Small program (Occam's Razor):

$$\sum_{i=1}^{n} i$$

Fast program (efficiency criteria):

$$\frac{n \times n + n}{2}$$

QSynt: Program Synthesis for Integer Sequences



Programming language

- Constants: $0, 1, 2$
- Variables: x, y
- Arithmetic: $+, -, \times, \mathit{div}, \mathit{mod}$
- Condition : if $\ldots \leqslant 0$ then $\ldots$ else $\ldots$
- $\mathit{loop}(f, a, b) := u_a$ where $u_0 = b$,

$$u_n = f(u_{n-1}, n)$$

- Two other loop constructs: *loop2*, a while loop

Example:

$$2^{\mathbf{x}} = \prod_{y=1}^{x} 2 = \mathit{loop}(2 \times x, \mathbf{x}, 1)$$

$$\mathbf{x}! = \prod_{y=1}^{x} y = \mathit{loop}(y \times x, \mathbf{x}, 1)$$

QSynt: Program Synthesis for Integer Sequences

- QSynt uses LLMs, Tree Neural Networks, and Monte Carlo Tree Search.
- It’s been looping for over a year without plateauing!
- 123k OEIS sequences (out of 350k) solved so far (600 iterations).

- You can try it here with your favorite sequences:

  http://grid01.ciirc.cvut.cz/~thibault/qsynt.html

- **Thank you :)**

- **If you’d like more,**
  - **My (former) PhD supervisor, Josef Urban, has recent slides introducing AITP: http://grid01.ciirc.cvut.cz/~mptp/sri24.pdf**
  - **I have research blog posts on my website that should be easier to digest than the papers: https://gardenofminds.art/research**