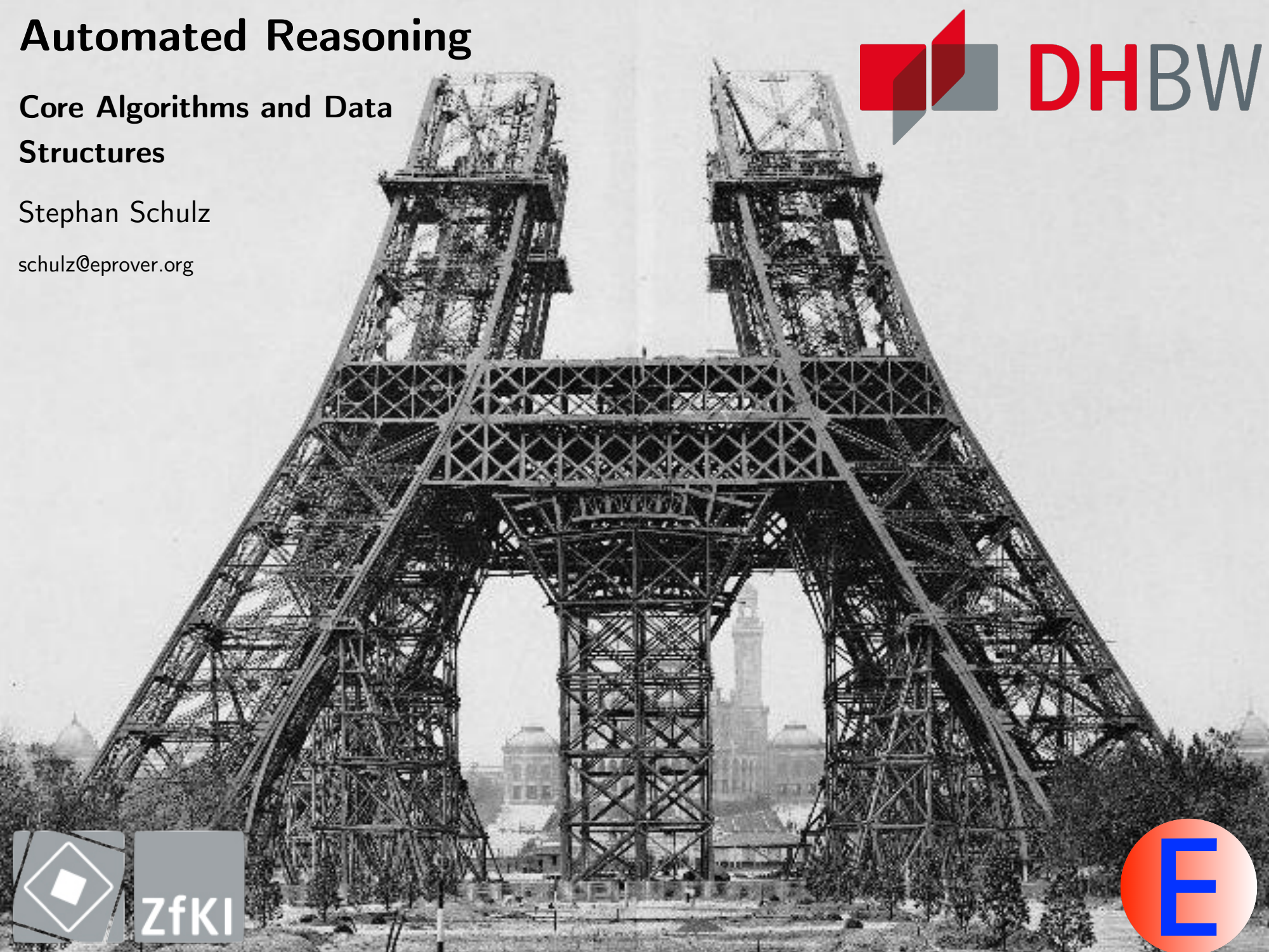
Automated Reasoning

Core Algorithms and Data Structures

Stephan Schulz

schulz@eprover.org



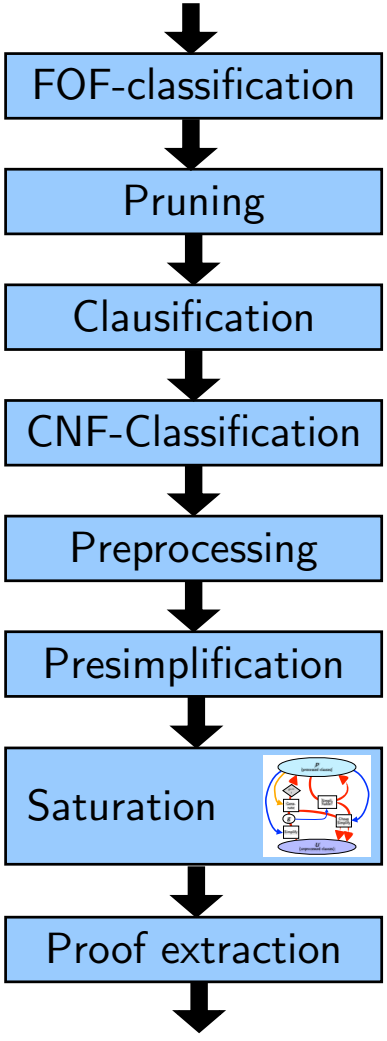
Abstract

We will look at some core concepts shared by nearly all modern automated theorem provers and related systems – terms, substitutions, unification, matching, as well as their applications, including paramodulation/superposition, rewriting, and subsumption. I will present actual code examples from PyRes and E, and discuss some implementation details.
We will then also attempt to develop a new, unification-based clause selection heuristic for E, and evaluate if it is a useful contribution to the portfolio of strategies.
Participants are encouraged to bring a laptop with a UNIX/Linux style C development system, and/or a recent Python-3 installation for the practical work.

Contents

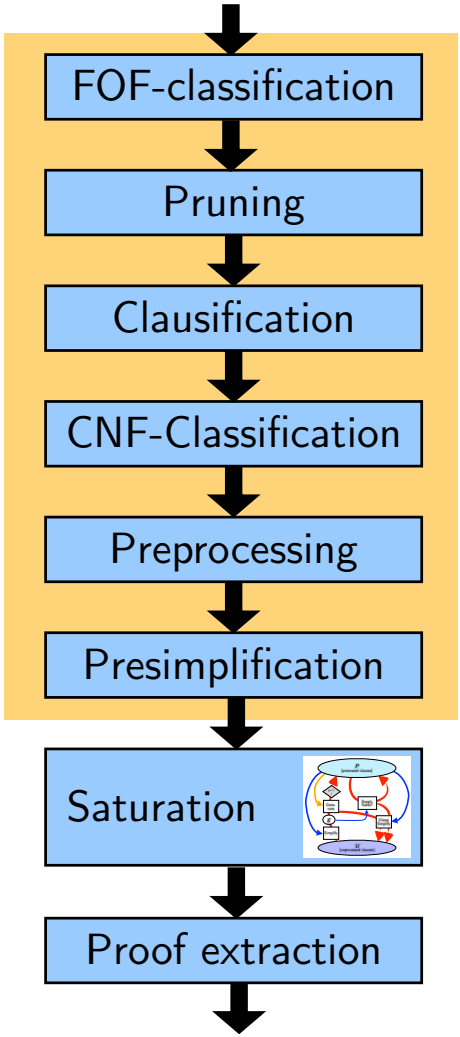
Introduction

Refutational Theorem Proving in First-Order Logic



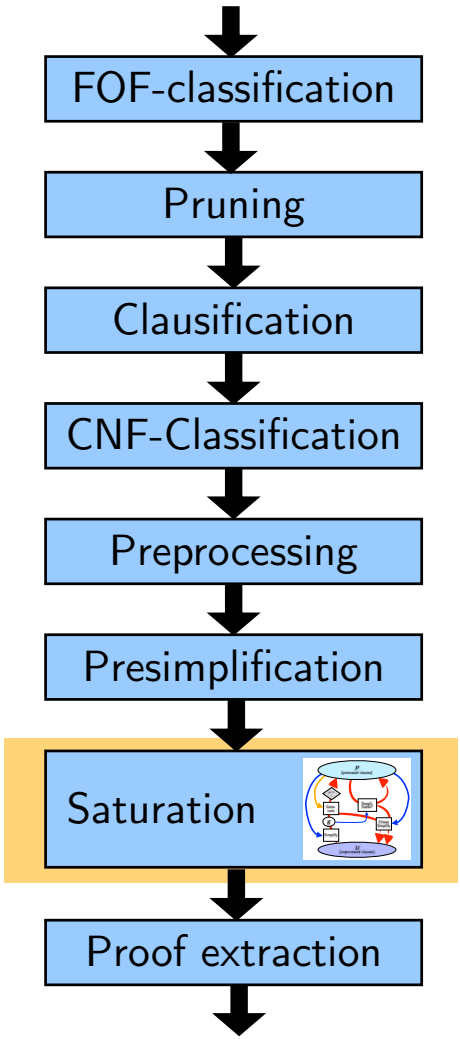
- High-level algorithm:

Refutational Theorem Proving in First-Order Logic



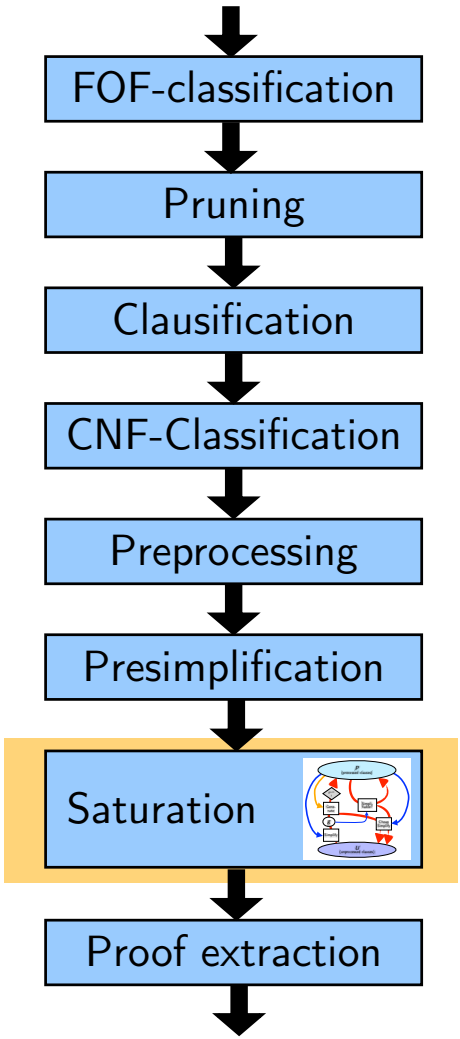
- High-level algorithm:
  - Do a lot of parsing and preprocessing
    - Result: A set of first-order clauses that is unsatisfiable if your conjecture holds

Refutational Theorem Proving in First-Order Logic



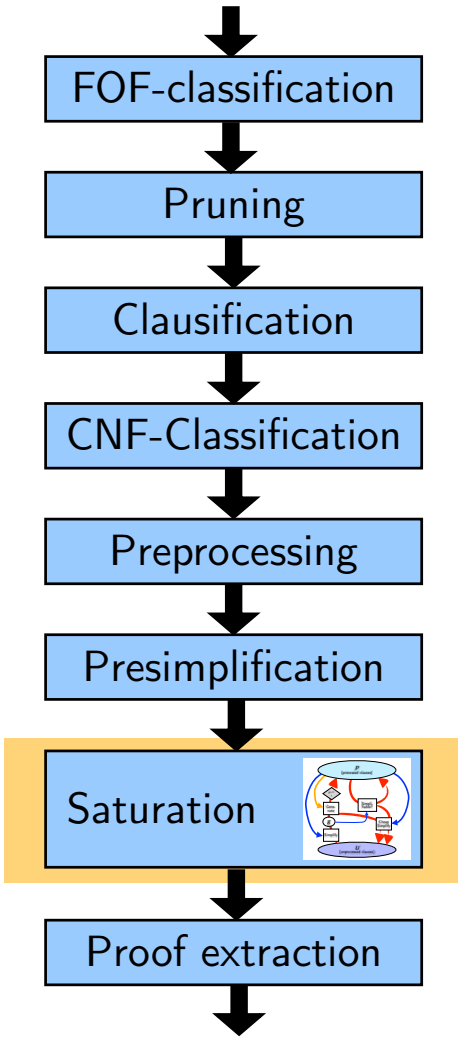
- High-level algorithm:
  - Do a lot of parsing and preprocessing
    - Result: A set of first-order clauses that is unsatisfiable if your conjecture holds
  - Saturate said clause set, trying to produce the empty clause as a witness of unsatisfiability

Refutational Theorem Proving in First-Order Logic



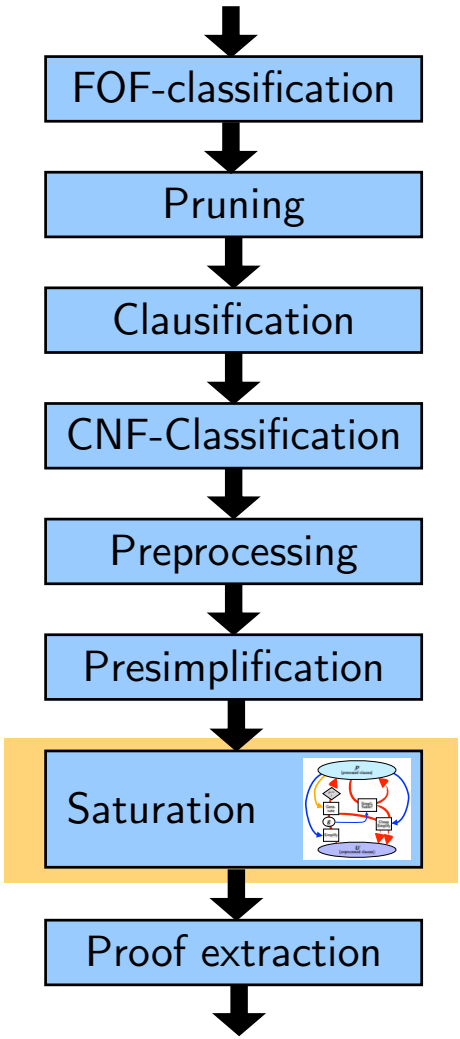
- High-level algorithm:
  - Do a lot of parsing and preprocessing
    - Result: A set of first-order clauses that is unsatisfiable if your conjecture holds
  - Saturate said clause set, trying to produce the empty clause as a witness of unsatisfiability
    1. Probably result: Timeout

Refutational Theorem Proving in First-Order Logic



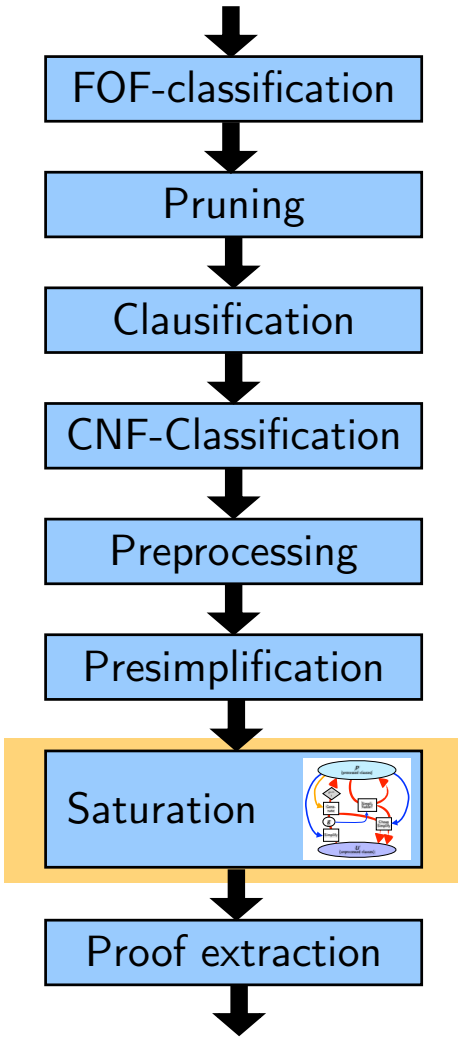
- High-level algorithm:
  - Do a lot of parsing and preprocessing
    - Result: A set of first-order clauses that is unsatisfiable if your conjecture holds
  - Saturate said clause set, trying to produce the empty clause as a witness of unsatisfiability
    1. Probably result: Timeout
    2. Possible result: Out of memory

Refutational Theorem Proving in First-Order Logic



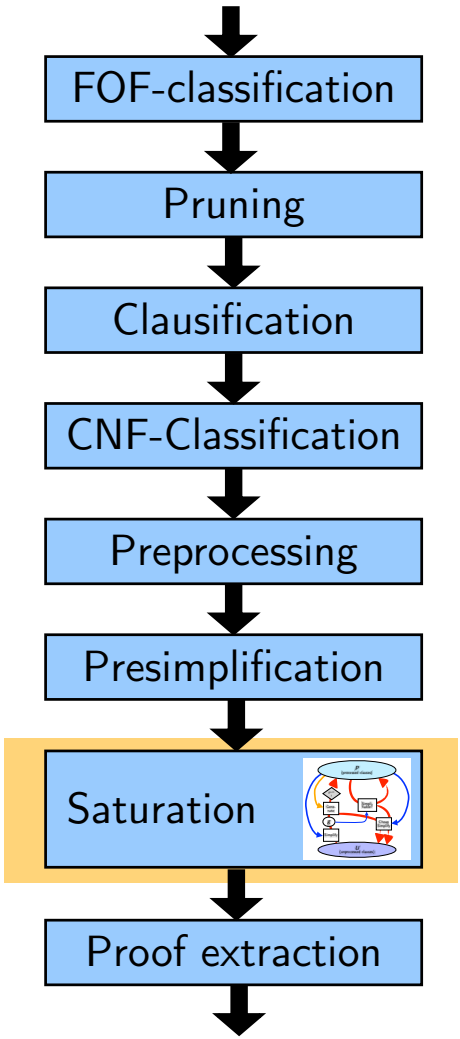
- High-level algorithm:
  - Do a lot of parsing and preprocessing
    - Result: A set of first-order clauses that is unsatisfiable if your conjecture holds
  - Saturate said clause set, trying to produce the empty clause as a witness of unsatisfiability
    1. Probably result: Timeout
    2. Possible result: Out of memory
    3. Occasional result: Incompleteness because you deleted critical clauses to avoid the previous case

Refutational Theorem Proving in First-Order Logic



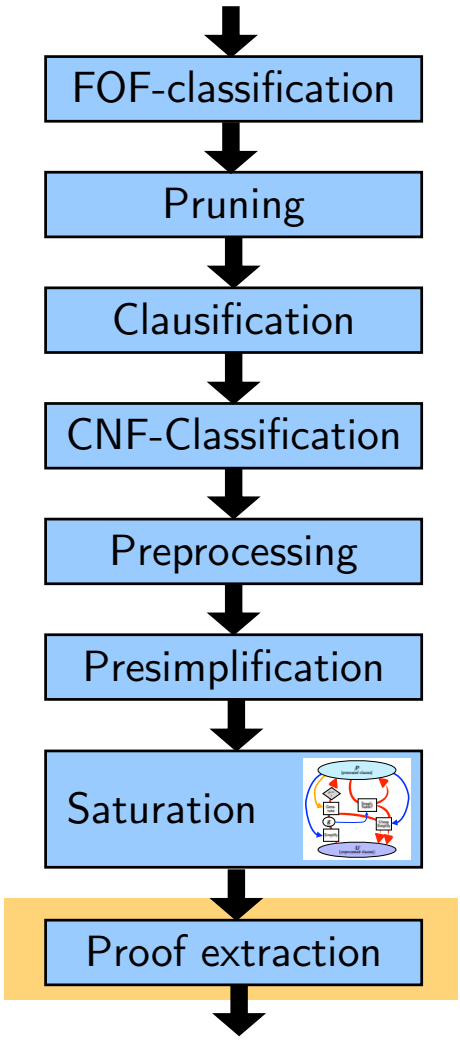
- High-level algorithm:
  - Do a lot of parsing and preprocessing
    - Result: A set of first-order clauses that is unsatisfiable if your conjecture holds
  - Saturate said clause set, trying to produce the empty clause as a witness of unsatisfiability
    1. Probably result: Timeout
    2. Possible result: Out of memory
    3. Occasional result: Incompleteness because you deleted critical clauses to avoid the previous case
    4. Rare result: Saturated clause set without empty clause (your conjecture does not hold!)

Refutational Theorem Proving in First-Order Logic



- High-level algorithm:
  - Do a lot of parsing and preprocessing
    - Result: A set of first-order clauses that is unsatisfiable if your conjecture holds
  - Saturate said clause set, trying to produce the empty clause as a witness of unsatisfiability
    1. Probably result: Timeout
    2. Possible result: Out of memory
    3. Occasional result: Incompleteness because you deleted critical clauses to avoid the previous case
    4. Rare result: Saturated clause set without empty clause (your conjecture does not hold!)
    5. Desired result: Empty clause

Refutational Theorem Proving in First-Order Logic



- High-level algorithm:
  - Do a lot of parsing and preprocessing
    - Result: A set of first-order clauses that is unsatisfiable if your conjecture holds
  - Saturate said clause set, trying to produce the empty clause as a witness of unsatisfiability
    1. Probably result: Timeout
    2. Possible result: Out of memory
    3. Occasional result: Incompleteness because you deleted critical clauses to avoid the previous case
    4. Rare result: Saturated clause set without empty clause (your conjecture does not hold!)
    5. Desired result: Empty clause
- Do a lot of post-processing
  - Result: Proof object

Saturation

- Assume a set of clauses C
- While the set is not saturated do...
  - Pick matching premises and an inference rule
  - Apply the inference rule to the premises and add the result to C
  - Perform simplification to remove redundant clauses from C

(Equality Resolution)

$$\frac{u \not\simeq v \vee R}{\sigma(R)} \quad \text{if } \sigma = \mathrm{mgu}(u, v), \ldots$$

(Deletion of resolved literals)

$$\frac{s \not\simeq s \vee R}{R}$$

(Superposition into negative literals)

$$\frac{s \simeq t \vee S \quad u \not\simeq v \vee R}{\sigma(u[p \leftarrow t] \not\simeq v \vee S \vee R)} \quad \text{if } \sigma = mgu(u|_p, s), \sigma(s) \not< \sigma(t), \ldots$$

(Rewriting of negative literals)

$$\frac{s \simeq t \quad u \not\simeq v \vee R}{s \simeq t \quad u[p \leftarrow \sigma(t)] \not\simeq v \vee R} \quad \text{if } u|_p = \sigma(s) \text{ and } \sigma(s) > \sigma(t)$$

(Equality Resolution)

$$\frac{u \not\simeq v \vee R}{\sigma(R)} \quad \text{if } \sigma = \mathrm{mgu}(u, v), \ldots$$

(Deletion of resolved literals)

$$\frac{s \not\simeq s \vee R}{R}$$

(Superposition into negative literals)

$$\frac{s \simeq t \vee S \quad u \not\simeq v \vee R}{\sigma(u[p \leftarrow t] \not\simeq v \vee S \vee R)} \quad \text{if } \sigma = mgu(u|_p, s), \sigma(s) \not< \sigma(t), \ldots$$

(Rewriting of negative literals)

$$\frac{s \simeq t \quad u \not\simeq v \vee R}{s \simeq t \quad u[p \leftarrow \sigma(t)] \not\simeq v \vee R} \quad \text{if } u|_p = \sigma(s) \text{ and } \sigma(s) > \sigma(t)$$

(Equality Resolution)

$$\frac{u \not\simeq v \vee R}{\sigma(R)} \quad \text{if } \sigma = \mathrm{mgu}(u, v), \ldots$$

Generating inference rules

- Necessary for completeness
- Increase size of proof state

(Superposition into negative literals)

$$\frac{s \simeq t \vee S \quad u \not\simeq v \vee R}{\sigma(u[p \leftarrow t] \not\simeq v \vee S \vee R)} \quad \text{if } \sigma = mgu(u|_p, s), \sigma(s) \not< \sigma(t), \ldots$$

(Deletion of resolved literals)

$$\frac{s \not\simeq s \vee R}{R}$$

Simplification rules

- Critical for performance
- Reduce size of proof state

(Rewriting of negative literals)

$$\frac{s \simeq t \quad u \not\simeq v \vee R}{s \simeq t \quad u[p \leftarrow \sigma(t)] \not\simeq v \vee R} \quad \text{if } u|_p = \sigma(s) \text{ and } \sigma(s) > \sigma(t)$$

(Equality Resolution)

$$\frac{u \not\simeq v \vee R}{\sigma(R)} \quad \text{if } \sigma = \mathrm{mgu}(u, v), \ldots$$

(Deletion of resolved literals)

$$\frac{s \not\simeq s \vee R}{R}$$

(Superposition into negative literals)

$$\frac{s \simeq t \vee S \quad u \not\simeq v \vee R}{\sigma(u[p \leftarrow t] \not\simeq v \vee S \vee R)} \quad \text{if } \sigma = mgu(u|_p, s), \sigma(s) \not< \sigma(t), \ldots$$

(Rewriting of negative literals)

$$\frac{s \simeq t \quad u \not\simeq v \vee R}{s \simeq t \quad u[p \leftarrow \sigma(t)] \not\simeq v \vee R} \quad \text{if } u|_p = \sigma(s) \text{ and } \sigma(s) > \sigma(t)$$

(Equality Resolution)

$$\frac{u \not\simeq v \vee R}{\sigma(R)} \quad \text{if } \sigma = \mathrm{mgu}(u, v), \ldots$$

(Deletion of resolved literals)

$$\frac{s \not\simeq s \vee R}{R}$$

(Superposition into negative literals)

$$\frac{s \simeq t \vee S \quad u \not\simeq v \vee R}{\sigma(u[p \leftarrow t] \not\simeq v \vee S \vee R)} \quad \text{if } \sigma = mgu(u|_p, s), \sigma(s) \not< \sigma(t), \ldots$$

(Rewriting of negative literals)

$$\frac{s \simeq t \quad u \not\simeq v \vee R}{s \simeq t \quad u[p \leftarrow \sigma(t)] \not\simeq v \vee R} \quad \text{if } u|_p = \sigma(s) \text{ and } \sigma(s) > \sigma(t)$$

(Equality Resolution)

$$\frac{u \not\simeq v \vee R}{\sigma(R)} \quad \text{if } \sigma = \mathrm{mgu}(u, v), \ldots$$

(Deletion of resolved literals)

$$\frac{s \not\simeq s \vee R}{R}$$

Local (single premise)

- Easy to keep track of
- Cheap to implement

Non-local (multiple premises)

- Harder to keep track of (pairs of clauses!)
- Expensive to implement (find partners)

(Superposition into negative literals)

$$\frac{s \simeq t \vee S \quad u \not\simeq v \vee R}{\sigma(u[p \leftarrow t] \not\simeq v \vee S \vee R)} \quad \text{if } \sigma = mgu(u|_p, s), \sigma(s) \not< \sigma(t), \ldots$$

(Rewriting of negative literals)

$$\frac{s \simeq t \quad u \not\simeq v \vee R}{s \simeq t \quad u[p \leftarrow \sigma(t)] \not\simeq v \vee R} \quad \text{if } u|_p = \sigma(s) \text{ and } \sigma(s) > \sigma(t)$$

(Equality Resolution)

$$\frac{u \not\simeq v \vee R}{\sigma(R)} \quad \text{if } \sigma = \mathrm{mgu}(u, v), \ldots$$

(Deletion of resolved literals)

$$\frac{s \not\simeq s \vee R}{R}$$

>99% of generated clauses

(Superposition into negative literals)

$$\frac{s \simeq t \vee S \quad u \not\simeq v \vee R}{\sigma(u[p \leftarrow t] \not\simeq v \vee S \vee R)} \quad \text{if } \sigma = mgu(u|_p, s), \sigma(s) \not< \sigma(t), \ldots$$

>90% of cpu time

(Rewriting of negative literals)

$$\frac{s \simeq t \quad u \not\simeq v \vee R}{s \simeq t \quad u[p \leftarrow \sigma(t)] \not\simeq v \vee R} \quad \text{if } u|_p = \sigma(s) \text{ and } \sigma(s) > \sigma(t)$$

(Equality Resolution)

$$\frac{u \not\simeq v \vee R}{\sigma(R)} \quad \text{if } \sigma = \mathrm{mgu}(u, v), \ldots$$

(Deletion of resolved literals)

$$\frac{s \not\simeq s \vee R}{R}$$

(Superposition into negative literals)

$$\frac{s \simeq t \vee S \quad u \not\simeq v \vee R}{\sigma(u[p \leftarrow t] \not\simeq v \vee S \vee R)} \quad \text{if } \sigma = mgu(u|_p, s), \sigma(s) \not< \sigma(t), \ldots$$

(Rewriting of negative literals)

$$\frac{s \simeq t \quad u \not\simeq v \vee R}{s \simeq t \quad u[p \leftarrow \sigma(t)] \not\simeq v \vee R} \quad \text{if } u|_p = \sigma(s) \text{ and } \sigma(s) > \sigma(t)$$

(Equality Resolution)

$$\frac{u \not\simeq v \vee R}{\sigma(R)} \quad \text{if } \sigma = \mathrm{mgu}(u, v), \ldots$$

(Deletion of resolved literals)

$$\frac{s \not\simeq s \vee R}{R}$$

(Superposition into negative literals)

$$\frac{s \simeq t \vee S \quad u \not\simeq v \vee R}{\sigma(u[p \leftarrow t] \not\simeq v \vee S \vee R)} \quad \text{if } \sigma = mgu(u|_p, s), \sigma(s) \not< \sigma(t), \ldots$$

(Rewriting of negative literals)

$$\frac{s \simeq t \quad u \not\simeq v \vee R}{s \simeq t \quad u[p \leftarrow \sigma(t)] \not\simeq v \vee R} \quad \text{if } u|_p = \sigma(s) \text{ and } \sigma(s) > \sigma(t)$$

(Equality Resolution)

$$\frac{u \not\simeq v \vee R}{\sigma(R)} \quad \text{if } \sigma = \mathrm{mgu}(u, v), \ldots$$

(Deletion of resolved literals)

$$\frac{s \not\simeq s \vee R}{R}$$

(Superposition into negative literals)

$$\frac{s \simeq t \vee S \quad u \not\simeq v \vee R}{\sigma(u[p \leftarrow t] \not\simeq v \vee S \vee R)} \quad \text{if } \sigma = mgu(u|_p, s), \sigma(s) \not< \sigma(t), \ldots$$

(Rewriting of negative literals)

$$\frac{s \simeq t \quad u \not\simeq v \vee R}{s \simeq t \quad u[p \leftarrow \sigma(t)] \not\simeq v \vee R} \quad \text{if } u|_p = \sigma(s) \text{ and } \sigma(s) > \sigma(t)$$

Basics

First-Order Terms

Definition (First-order terms)

We assume a set F of function symbols with associated arities (e.g. $\{f/2, g/1, a/0, b/9\}$) and a set of first-order variables $V = \{X, Y, Z, \ldots\}$. The set $T(F, V)$ of terms is defined as follows:

- $V \subseteq T(F, V)$ (any variable is a term)
- If $t_1, \ldots, t_n$ are in $T(F, V)$ and $f/n \in F$, then $f(t_1, \ldots, t_n) \in T(F, V)$ (an n-ary function symbol with n argument terms is a term)
- $T(V, F)$ is the smallest set fulfilling these conditions

First-Order Terms

Definition (First-order terms)

We assume a set F of function symbols with associated arities (e.g. $\{f/2, g/1, a/0, b/9\}$) and a set of first-order variables $V = \{X, Y, Z, \ldots\}$. The set $T(F, V)$ of terms is defined as follows:

- $V \subseteq T(F, V)$ (any variable is a term)
- If $t_1, \ldots, t_n$ are in $T(F, V)$ and $f/n \in F$, then $f(t_1, \ldots, t_n) \in T(F, V)$ (an n-ary function symbol with n argument terms is a term)
- $T(V, F)$ is the smallest set fulfilling these conditions

- Examples (over the F, V given above):
  - $f(X, g(Y))$
  - $a()$ (usually written as a without parentheses - a is a constant)
  - $g(g(g(f(Y, b))))$

Substitutions

Definition (Substitution)

- A substitution is a function $\sigma : V \to T(F, V)$ mapping variables to terms with $dom(\sigma) = \{X \in V \mid |\sigma(X) \neq x\}$ is finite.

Substitutions

Definition (Substitution)

- A substitution is a function $\sigma : V \to T(F, V)$ mapping variables to terms with $dom(\sigma) = \{X \in V \mid |\sigma(X) \neq x\}$ is finite.
- If $C \in dom(\sigma)$, we say "X is bound (to $\sigma(X)$) by σ".
- We extend substitutions in the obvious way to functions we can apply to terms, atoms, literals and clauses.
- We write e.g. $\sigma = \{X \mapsto a, Y \mapsto f(a, X), Z \mapsto g(g(b))\}$

Substitutions allow us to systematically replace variables by terms:

Substitutions

Definition (Substitution)

- A substitution is a function $\sigma : V \to T(F, V)$ mapping variables to terms with $dom(\sigma) = \{X \in V \mid |\sigma(X) \neq x\}$ is finite.
- If $C \in dom(\sigma)$, we say "X is bound (to $\sigma(X)$) by σ".
- We extend substitutions in the obvious way to functions we can apply to terms, atoms, literals and clauses.
- We write e.g. $\sigma = \{X \mapsto a, Y \mapsto f(a, X), Z \mapsto g(g(b))\}$

Substitutions allow us to systematically replace variables by terms:

- Example: Consider $\sigma = \{X \mapsto a, Y \mapsto f(X, b)\}$
  - $\sigma(X) =$

Substitutions

Definition (Substitution)

- A substitution is a function $\sigma : V \to T(F, V)$ mapping variables to terms with $dom(\sigma) = \{X \in V \mid |\sigma(X) \neq x\}$ is finite.
- If $C \in dom(\sigma)$, we say "X is bound (to $\sigma(X)$) by σ".
- We extend substitutions in the obvious way to functions we can apply to terms, atoms, literals and clauses.
- We write e.g. $\sigma = \{X \mapsto a, Y \mapsto f(a, X), Z \mapsto g(g(b))\}$

Substitutions allow us to systematically replace variables by terms:

- Example: Consider $\sigma = \{X \mapsto a, Y \mapsto f(X, b)\}$
  - $\sigma(X) = a$

Substitutions

Definition (Substitution)

- A substitution is a function $\sigma : V \to T(F, V)$ mapping variables to terms with $dom(\sigma) = \{X \in V \mid |\sigma(X) \neq x\}$ is finite.
- If $C \in dom(\sigma)$, we say "X is bound (to $\sigma(X)$) by σ".
- We extend substitutions in the obvious way to functions we can apply to terms, atoms, literals and clauses.
- We write e.g. $\sigma = \{X \mapsto a, Y \mapsto f(a, X), Z \mapsto g(g(b))\}$

Substitutions allow us to systematically replace variables by terms:

- Example: Consider $\sigma = \{X \mapsto a, Y \mapsto f(X, b)\}$
  - $\sigma(X) = a$
  - $\sigma(\sigma(g(Y))) =$

Substitutions

Definition (Substitution)

- A substitution is a function $\sigma : V \to T(F, V)$ mapping variables to terms with $dom(\sigma) = \{X \in V \mid |\sigma(X) \neq x\}$ is finite.
- If $C \in dom(\sigma)$, we say "X is bound (to $\sigma(X)$) by σ".
- We extend substitutions in the obvious way to functions we can apply to terms, atoms, literals and clauses.
- We write e.g. $\sigma = \{X \mapsto a, Y \mapsto f(a, X), Z \mapsto g(g(b))\}$

Substitutions allow us to systematically replace variables by terms:

- Example: Consider $\sigma = \{X \mapsto a, Y \mapsto f(X, b)\}$
  - $\sigma(X) = a$
  - $\sigma(\sigma(g(Y))) = \sigma(g(f(X, b)))$

Substitutions

Definition (Substitution)

- A substitution is a function $\sigma : V \to T(F, V)$ mapping variables to terms with $dom(\sigma) = \{X \in V \mid |\sigma(X) \neq x\}$ is finite.
- If $C \in dom(\sigma)$, we say "X is bound (to $\sigma(X)$) by σ".
- We extend substitutions in the obvious way to functions we can apply to terms, atoms, literals and clauses.
- We write e.g. $\sigma = \{X \mapsto a, Y \mapsto f(a, X), Z \mapsto g(g(b))\}$

Substitutions allow us to systematically replace variables by terms:

- Example: Consider $\sigma = \{X \mapsto a, Y \mapsto f(X, b)\}$
  - $\sigma(X) = a$
  - $\sigma(\sigma(g(Y))) = \sigma(g(f(X, b))) = g(f(a, b))$

Substitutions

Definition (Substitution)

- A substitution is a function $\sigma : V \to T(F, V)$ mapping variables to terms with $dom(\sigma) = \{X \in V \mid |\sigma(X) \neq x\}$ is finite.
- If $C \in dom(\sigma)$, we say "X is bound (to $\sigma(X)$) by σ".
- We extend substitutions in the obvious way to functions we can apply to terms, atoms, literals and clauses.
- We write e.g. $\sigma = \{X \mapsto a, Y \mapsto f(a, X), Z \mapsto g(g(b))\}$

Substitutions allow us to systematically replace variables by terms:

- Example: Consider $\sigma = \{X \mapsto a, Y \mapsto f(X, b)\}$
  - $\sigma(X) = a$
  - $\sigma(\sigma(g(Y))) = \sigma(g(f(X, b))) = g(f(a, b))$
  - $\sigma(X \simeq Y \vee g(Y) \not\simeq a) =$

Substitutions

Definition (Substitution)

- A substitution is a function $\sigma : V \to T(F, V)$ mapping variables to terms with $dom(\sigma) = \{X \in V \mid |\sigma(X) \neq x\}$ is finite.
- If $C \in dom(\sigma)$, we say "X is bound (to $\sigma(X)$) by σ".
- We extend substitutions in the obvious way to functions we can apply to terms, atoms, literals and clauses.
- We write e.g. $\sigma = \{X \mapsto a, Y \mapsto f(a, X), Z \mapsto g(g(b))\}$

Substitutions allow us to systematically replace variables by terms:

- Example: Consider $\sigma = \{X \mapsto a, Y \mapsto f(X, b)\}$
  - $\sigma(X) = a$
  - $\sigma(\sigma(g(Y))) = \sigma(g(f(X, b))) = g(f(a, b))$
  - $\sigma(X \simeq Y \vee g(Y) \not\simeq a) = a \simeq f(X, b) \vee g(f(X, b)) \not\simeq a$

Composition of substitution

Definition (Composition of substitutions)

Let σ, τ be two substitutions

- $\tau \circ \sigma$ is a substitution with $\sigma \circ \tau(t) = \sigma(\tau(t))$ for all terms t

Substitutions are functions, functions can be composed.

Composition of substitution

Definition (Composition of substitutions)

Let σ, τ be two substitutions

- $\tau \circ \sigma$ is a substitution with $\sigma \circ \tau(t) = \sigma(\tau(t))$ for all terms t

Substitutions are functions, functions can be composed.

We can compute $\tau \circ \sigma$ explicitly:

$$\tau \circ \sigma = \{X \mapsto \tau(\sigma(X)) \mid X \text{ is bound by } \sigma\}$$
$$\cup \{X \mapsto \tau(X) \mid X \text{ is not bound by } \sigma\}$$

Composition of substitution

Definition (Composition of substitutions)

Let σ, τ be two substitutions

- $\tau \circ \sigma$ is a substitution with $\sigma \circ \tau(t) = \sigma(\tau(t))$ for all terms t

Substitutions are functions, functions can be composed.

We can compute $\tau \circ \sigma$ explicitly:

$$\begin{aligned}\tau \circ \sigma = \{X \mapsto \tau(\sigma(X)) \mid X \text{ is bound by } \sigma\} \\ \cup \{X \mapsto \tau(X) \mid X \text{ is not bound by } \sigma\}\end{aligned}$$

Special case: τ is of the form $\{X \mapsto t\}$ and X is not bound by σ:

$$\tau \circ \sigma = \{X \mapsto \tau(\sigma(X)) \mid X \text{ is bound by } \sigma\} \cup \tau$$

Unifiers

Definition (Unifier, most general unifier)

Let s, t be two terms.
A unifier for s, t...

- is a substitution σ
- such that $\sigma(s) = \sigma(t)$.

A unifier σ for s, t is called a most general unifier ...

- if every other unifier σ' of s, t can be written as $\tau \circ \sigma$ for some substitution τ

Unifiers

Definition (Unifier, most general unifier)

Let s, t be two terms.
A unifier for s, t...

- is a substitution σ
- such that $\sigma(s) = \sigma(t)$.

A unifier σ for s, t is called a most general unifier ...

- if every other unifier σ' of s, t can be written as $\tau \circ \sigma$ for some substitution τ

- Fact: If any unifier exists, then there is a (except for variable renaming) unique *most general unifier*
  - We therefore write $\sigma = mgu(s, t)$ and call it the MGU.

Unifiers

Definition (Unifier, most general unifier)

Let s, t be two terms.
A unifier for s, t...

- is a substitution σ
- such that $\sigma(s) = \sigma(t)$.

A unifier σ for s, t is called a most general unifier ...

- if every other unifier σ' of s, t can be written as $\tau \circ \sigma$ for some substitution τ

- Fact: If any unifier exists, then there is a (except for variable renaming) unique *most general unifier*
  - We therefore write $\sigma = mgu(s, t)$ and call it the MGU.
- Fact: MGUs can be found systematically

Why is unification interesting?

- *The calculus says I need unifiers!*

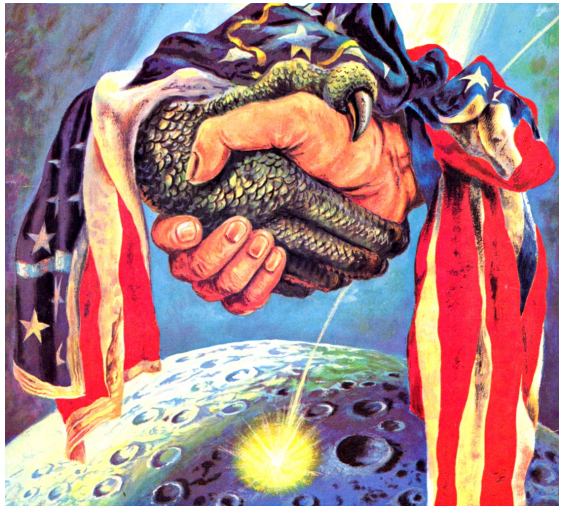
Why is unification interesting?

- *The calculus says I need unifiers!*
- But why?
  - Terms represent sets of objects
  - Atomic formulas are statements about sets of objects
  - . . . and so are clauses
- To usefully combine different knowledge fragments, they need to talk about a common set of objects
  - Unification determines if such a set exists. . .
  - . . . and in the success case gives us a description of this common set

Unification finds common instances of terms/atoms/clauses, i.e. the intersection of the domains they make useful statements about!

Observations on unification

1. $abs(X), dir(X)$ can never be unified
   - The first symbol is always different
2. $X, sqrt(X)$ can never be unified
   - No matter what is used for X, a *sqrt* always remains
   - "Occurs-Check"
3. $X, sqrt(a)$ unifies with $\sigma = \{X \mapsto sqrt(a)\}$
   - A variable paired with most terms is fine
4. $add(X, a), add(b, Y)$ unifies with $\sigma = \{X \mapsto b, Y \mapsto a\}$
   - The unifier can be composed from those necessary to unify particular subterms



Unification as parallel equation solving

Fact: The unification problem becomes simpler when you consider it for sets of pairs of terms!

- Given: $C = \{s_1 = t_1, s_2 = t_2, \ldots, s_n = t_n\}$
  - Search common mgu σ with
    - $\sigma(s_1) = \sigma(t_1)$
    - $\sigma(s_2) = \sigma(t_2)$
    - $\ldots$
    - $\sigma(s_n) = \sigma(t_n)$
- Use a transformation system ([LMM88])
  - State: C, σ
    - C: Set of equations to be solved
    - σ: Candidate unifier
  - Initial state for finding mgu(s, t): $\{s = t\}, \{\}$
  - Termination: $\{\}, \sigma$

Unification: Transformation system

$$\text{Deletion:}\quad \frac{\{t = t\} \cup C, \sigma}{C, \sigma}$$

$$\text{Bind:}\quad \frac{\{X = t\} \cup C, \sigma}{\{X \mapsto t\}(C), \{X \mapsto t\} \circ \sigma} \text{ if } X \notin \text{Vars}(t)$$

$$\text{Orient:}\quad \frac{\{t = X\} \cup C, \sigma}{\{X = t\} \cup C, \sigma} \text{ if } t \text{ is not a variable}$$

$$\text{Decompose:}\quad \frac{\{f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)\} \cup C, \sigma}{\{s_1 = t_1, \ldots, s_n = t_n\} \cup C, \sigma}$$

$$\text{Occurs:}\quad \frac{\{X = t\} \cup C, \sigma}{\mathit{FAIL}} \text{ if } X \in \text{Vars}(t), t \neq X$$

$$\text{Conflict:}\quad \frac{\{f(s_1, \ldots, s_n) = g(t_1, \ldots, t_m)\} \cup C, \sigma}{\mathit{FAIL}} \text{ if } f \neq g$$

Unification algorithm (sketch)

```
def mgu(s,t):
    C={s=t}
    sigma = {}
    while C != {}:
        remove arbitrary s=t from C
        apply the matching rule to s=t, C, sigma
            (modifying sigma and C as appropriate)
        if result==FAIL:
            return FAIL
    return sigma
```

Example

C	σ	Rule
$\{f(f(X, g(g(Y))), X) = f(f(Z, Z), U)\}$	$\{\}$	Decompose
$\{f(X, g(g(Y))) = f(Z, Z), X = U\}$	$\{\}$	Bind (X)
$\{f(U, g(g(Y))) = f(Z, Z)\}$	$\{X \mapsto U\}$	Decompose
$\{U = Z, g(g(Y))) = Z\}$	$\{X \mapsto U\}$	Bind (U)
$\{g(g(Y)) = Z\}$	$\{X \mapsto Z,$ $U \mapsto Z\}$	Orient
$\{Z = g(g(Y))\}$	$\{X \mapsto Z,$ $U \mapsto Z\}$	Bind (Z)
$\{\}$	$\{X \mapsto g(g(Y)),$ $U \mapsto g(g(Y)),$ $Z \mapsto g(g(Y))\}$	

Pretty Code

Python: Terms

- Terms are implemented as Python lists (equivalent to LISP s-expressions)
  - Variables are plain strings: "X", "Y", ...
  - Function symbols are plain strings: "f", "g", "sqrt", ...
  - Composite terms: [f, $t_1, \ldots, t_n$]
    - f is the function symbol
    - The t_i represent the subterms
- Example: $f(f(X, a), g(a))$
  - Python representation: ["f", ["f", "X", ["a"]], ["g" ["a"]]]
- Functions include:
  - `termIsVar(t)` - return true if `t` is a variable
  - `termIsCompound(t)` - return true if `t` is not a variable
  - `termFunc(t)` - return the function symbol of compound term `t`
  - `termArgs(t)` - return the argument list of compound term `t`

Python substitutions (basics)

```
class Substitution(object):
    def __init__(self, init = []):
        self.subst = {}
        for i in init:
            self.subst[i[0]]=i[1]

    def value(self, var):
        if var in self.subst:
            return self.subst[var]
        else:
            return var

    def apply(self, term):
        if termIsVar(term):
            return self.value(term)
        else:
            res  = [termFunc(t)]
            args = [self.apply(x) for x in termArgs(term)]
            res.extend(args)
            return res
```

Python substitutions (extension)

```
def isBound(self, var):
    return var in self.subst

def modifyBinding(self, binding):
    var, term = binding
    if self.isBound(var):
        res = self.value(var)
    else:
        res = None
    if term == None:
        if self.isBound(var):
            del self.subst[var]
    else:
        self.subst[var] = term
    return res
```

Python substitutions (limited composition)

```
def composeBinding(self, binding):
    tmpsubst = Substitution([binding])
    var, term = binding
    vars = self.subst.keys()
    for x in vars:
        bound = self.subst[x]
        self.subst[x] = tmpsubst.apply(bound)
    if not var in self.subst:
        self.subst[var] = term
```

Python: Occurs check

```
def occursCheck(x, t):
    """
---Perform an occurs-check, i.e. determine if the variable
---x occurs in the term t. If that is the case (and
---t != x), the two can never be unified.
---"""
    if termIsCompound(t):
        for i in termArgs(t):
            if occursCheck(x, i):
                return True
        return False
    else:
        return x == t
```

- Simple recursive descent

Python Unification

```
def mguTermList(l1, l2, subst):
    while(len(l1)!=0):
        t1 = l1.pop(0)
        t2 = l2.pop(0)
        if termIsVar(t1):
            if t1==t2:
                continue
            if occursCheck(t1, t2):
                return None
            new_binding = Substitution([(t1, t2)])
            l1 = [new_binding(t) for t in l1]
            l2 = [new_binding(t) for t in l2]
            subst.composeBinding((t1, t2))
        elif termIsVar(t2):
            [symmetric case elided]
        else:
            if termFunc(t1) != termFunc(t2):
                return None
            l1.extend(termArgs(t1))
            l2.extend(termArgs(t2))
    return subst
```

Hacking Time

Beyond unification

- For heuristic evaluation, we want to go beyond unification success/failure
- We want a measure of "how close" two terms are to being unifiable
- Idea: Compute a unification distance
  - For unifiable term pairs the distance is 0
  - Otherwise, collect all unsolved equations
    - Compute numerical score from these unsolved equations
    - ...e.g. summing up the size of all terms involved
- Example: Trying to unify $f(g(a), f(X, a))$ and $f(g(b), f(g(X), Y))$
  - Repeated decomposition yields $a = b, X = g(X), a = Y$
  - The green equation is solvable, the red ones yield a unification distance of (1+1)+(1+2)=5

Exercise: Implement unification distance

- Implement unification distance in PyRes
- Suggestions:
  - Use the file `unification.py`
  - Steal what you can (we have limited time)
  - Integrate test code into the unit tests at the end of the file
  - To run the unit tests, just run the file (`./unification.py`)
  - What is the unification distance of $p(a, b, c, X, Y)$ and $p(b, c, d, f(a), f(Y))$?
- Reminder: `git clone`
  `https://github.com/eprover/PyRes.git`
- To run an example:
  - `cd PyRes`
  - `./pyres-fof.py -tifbp -HPickGiven5 -nlargest EXAMPLES/PUZ001+1.p`

A note on matching

- *Matching* is the task of finding a substitution σ with $\sigma(s) = t$
  - Substitution is only applied to one term (the matching term)
- Differences:
  - No occurs-check necessary
  - A binding once made is invariant
  - In particular: No composition of bindings
  - $X = X$ cannot be discarded, but must be bound
- Result: Matching is in $\Theta(n)$
  - One linear pass is enough
  - Check `matching.py` for details in PyRes

Performance Code

PyRes vs. Performance

- PyRes is written with the aim of clarity
  - Performance is a secondary consideration
  - Python is not a high-performance language
- Can we do better?
  - Algorithmically?
  - Data-structure-wise?
  - Choosing a different language?

PyRes vs. Performance

- PyRes is written with the aim of clarity
  - Performance is a secondary consideration
  - Python is not a high-performance language
- Can we do better?
  - Algorithmically?
  - Data-structure-wise?
  - Choosing a different language?

Yes, yes, and yes!

Observations

- In practice, nearly all unification attempts fail!
  - No paramodulation into/resolution on first-order variables
  - 10 different function symbols $\Rightarrow\approx$ 90% conflict on the first symbol
- Efficient unification: Detect failures early
  - Prefer equations matching Decompose and Conflict
  - Delay equations matching Orient, Bind, Occurs
  - Deletion is only needed for equations of the form $X = X$ (Why?)

Data types in C

From `E/cte_termtypes.h`:

```
typedef struct termcell
{
   FunCode           f_code;     /* Top symbol of term */
   TermProperties    properties; /* Boolean flags */
   int               arity;
   struct termcell*  binding;    /* For variable bindings,
                                    potentially for temporary
                                    rewrites */
   [...a lot of members elided...]
   struct termcell*  args[];     /* Flexible array member contain
                                    the arguments */
}TermCell, *Term_p, **TermRef;
```

- Function symbol/variable encoded as `FunCode` (i.e. `long`)
- Arguments as flexible array member

Data types in C

From `E/cte_termtypes.h`:

```
typedef struct termcell
{
   FunCode           f_code;      /* Top symbol of term */
   TermProperties    properties;  /* Boolean flags */
   int               arity;
   struct termcell*  binding;     /* For variable bindings,
                                     potentially for temporary
                                     rewrites */
   [...a lot of members elided...]
   struct termcell*  args[];      /* Flexible array member contain
                                     the arguments */
}TermCell, *Term_p, **TermRef;
```

- Function symbol/variable encoded as `FunCode` (i.e. `long`)
- Arguments as flexible array member
- Variable binding directly in the term cell

Shared terms and shared variable bindings

- E aggressively shares terms [Sch25]
  - Every long-lived term is represented only once
  - Variables are even more aggressively shared - every variable corresponds to exactly one term cell (even in unshared terms)
  - Originally inspired by DeDam [NRV97], although the relationship is hard to recognise now...
- What does that mean?
  - If we bind a variable anywhere, we bind a variable everywhere
  - Some application of substitutions to complex structures can be done locally (if we have the variables in hand)

Substitutions in E

```
typedef PStack_p    Subst_p;
```

Substitutions in E

```
typedef PStack_p    Subst_p;
```

- A substitution is just a stack (of term cell pointers)
  - When a variable is bound, the binding is recorded in the term cell
  - The fact that this binding is part of a substitution is recorded by pushing the variable onto the substitution
  - In other words, a single binding is the pair `(v, v->binding)`
  - Each bound variable (representing the binding) is on the stack
- Allows for fast backtracking
- Since variables are shared, adding a binding is the same as composing a new binding with the substitution!
- Big performance gain
  - Intermediate results are represented implicitly
  - Deleting them is nearly free

Backtracking substitutions

```
PStackPointer SubstAddBinding(Subst_p subst, Term_p var,
                              Term_p bind)
{
   PStackPointer ret = PStackGetSP(subst);
   var->binding = bind;
   PStackPushP(subst, var);
   return ret;
}

bool SubstBacktrackSingle(Subst_p subst)
{
   Term_p handle;
   if(PStackEmpty(subst))
   {
      return false;
   }
   handle = PStackPopP(subst);
   handle->binding = NULL;
   return true;
}
```

Substitution code

```
int SubstBacktrackToPos(Subst_p subst, PStackPointer pos)
{
    int ret = 0;
    while(PStackGetSP(subst) > pos)
    {
        SubstBacktrackSingle(subst);
        ret++;
    }
    return ret;
}

int SubstBacktrack(Subst_p subst)
{
    int ret = 0;
    while(SubstBacktrackSingle(subst))
    {
        ret++;
    }
    return ret;
}
```

Unification code in E - Prologue

```
bool SubstComputeMgu(Term_p t1, Term_p t2, Subst_p subst)
{
   if((TermCellQueryProp(t1, TPPredPos) && TermIsFreeVar(t2))||
      (TermCellQueryProp(t2, TPPredPos) && TermIsFreeVar(t1)))
   {
      return false;

   }
   PStackPointer backtrack = PStackGetSP(subst); /* For backtra

   bool res = true;
   PQueue_p jobs = PQueueAlloc();

   PQueueStoreP(jobs, t1);
   PQueueStoreP(jobs, t2);
```

Unification code in E - Orient, Bind

```
while(!PQueueEmpty(jobs))
{
   t2 =  TermDerefAlways(PQueueGetLastP(jobs));
   t1 =  TermDerefAlways(PQueueGetLastP(jobs));
   if(TermIsFreeVar(t2))
   {
      SWAP(t1, t2);
   }
   if(TermIsFreeVar(t1))
   {
      if(t1 != t2)
      {
         if((t1->type != t2->type) || OccurCheck(t2, t1))
         {
            res = false;
            break;
         }
         else
         {
            SubstAddBinding(subst, t1, t2);
   }  }  }
```

Unification code in E - Conflict

```
else
{
    if(t1->f_code != t2->f_code)
    {
        res = false;
        break;
    }
    else
    {
```

Unification code in E - Decompose

```
                for(int i=t1->arity-1; i>=0; i--)
                {
                   /* Delay variable bindings */
                   if(TermIsFreeVar(t1->args[i]) ||
                      TermIsFreeVar(t2->args[i]))
                   {
                      PQueueBuryP(jobs, t2->args[i]);
                      PQueueBuryP(jobs, t1->args[i]);
                   }
                   else
                   {
                      PQueueStoreP(jobs, t1->args[i]);
                      PQueueStoreP(jobs, t2->args[i]);
                   }
                }
             }
          }
       }
```

Unification code in E - Epilogue

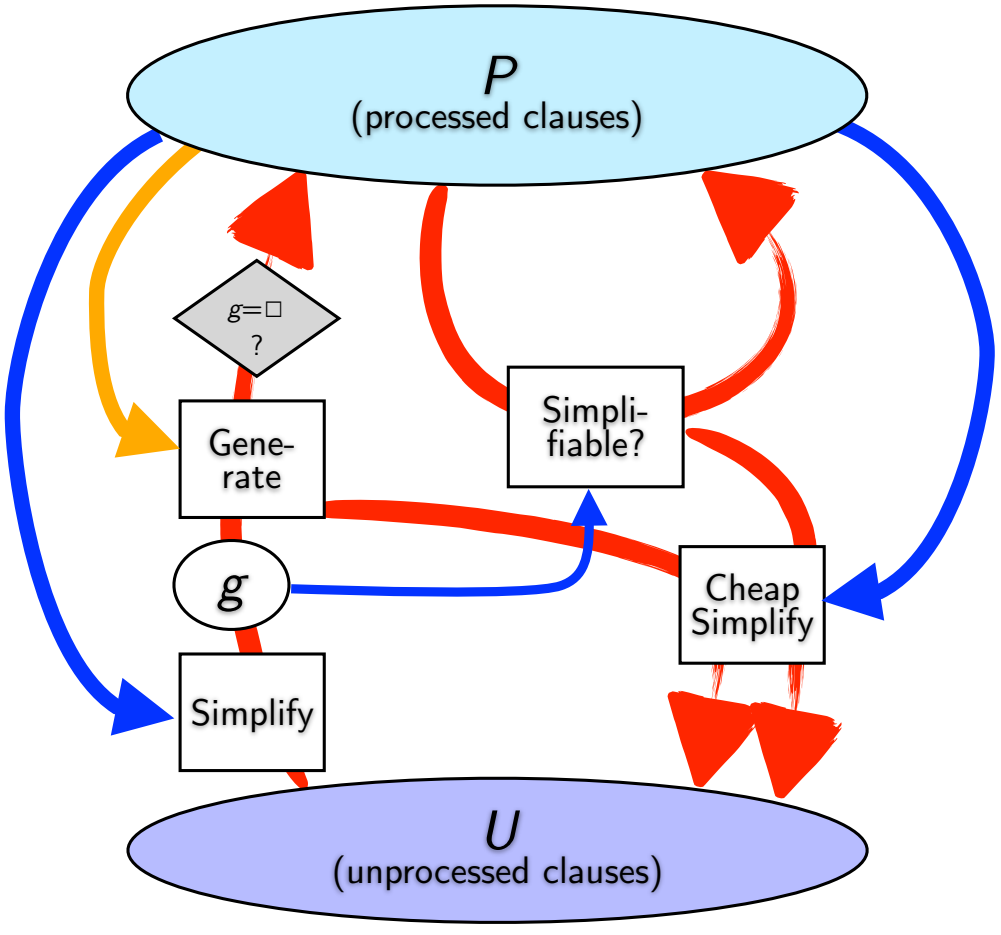
```
    PQueueFree(jobs);

    if(!res)
    {
        SubstBacktrackToPos(subst,backtrack);
    }
    return res;
}
```

(see E/TERMS/cte_match_mgu_1-1.[ch])

Using Unification Distance

Given Clause Loop



- While there are unprocessed clauses . . .
  - . . . heuristically pick given clause from U
  - . . . and process it
- Goal: Generate empty clause
- Choice point: Pick the next given clause
  - Based on heuristic weight (small is beautiful)

Unification distance in clause evaluation

- We are in an equational context:
  - Literals are equations or disequations
  - Goal: Empty clause (get rid of all literals)
- Consider $a \not\simeq X \vee f(X, a) \simeq a$
  - Implicational form: $a \simeq X \to f(X, a) \simeq a$
  - So: If we can solve the first literal, the second becomes available for rewriting
  - So: Negative literals that can be resolved (by unification) are desirable
  - Maybe: Negative literals that can be nearly resolved are too?
- Idea: Base literal evaluation on unification distance
  - Sum them up (cleverly) for clauses
  - Cleverly: What do we do for positive literals?

Hacking session 2

- Reminder: E lives at `https://www.eprover.org`
- ...or at `https://github.com/eprover/eprover`
  - ...which also displays the installation information
- Option 1:
  - Improve the PyRes unification algorithm
  - Steal ideas from the C version
- Option 2:
  - Implement unification distance in E
  - Use it to implement a new clause evaluation heuristic

Clause evaluation in E

- All relevant code is in `E/HEURISTICS/`
- Each clause evaluation function needs 3 C functions:
  - `WFCB_p MyEvalParse((Scanner_p in, OCB_p ocb, ProofState_p state)`
  - `WFCB_p MyEvalInit(ClausePrioFun prio_fun, ...)` (which will only be called from `MyEvalParse()`
  -
  - `double MyEvalCompute(void* data, Clause_p clause)` (which will be called for each clause)
- You need to register your new function in `che_wfcbadmin.c`
  - External name goes into `WeightFunParseFunNames[]`
  - Parse function goes into `WeightFunParseFun[]` (at the corresponding index)
- Check e.g. `TPTPTypeWeight*` in `HEURISTICS/che_varweights.h` for a simple example

Conclusion

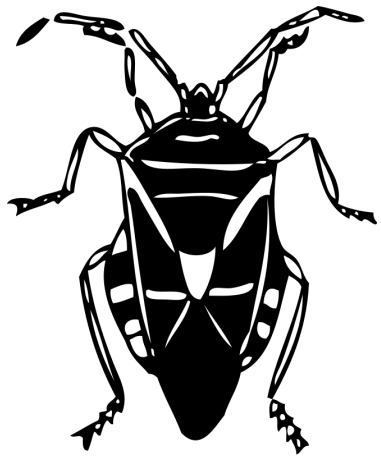
- First steps into some data structures
  - Terms
  - Substitutions
- Unification as equation solving . . .
  - . . . not as a theoretical approach, but as a practical implementation
- Some hacking experience with real ATPs gained(?!?)

Conclusion

- First steps into some data structures
  - Terms
  - Substitutions
- Unification as equation solving . . .
  - . . . not as a theoretical approach, but as a practical implementation
- Some hacking experience with real ATPs gained(?!?)

I hope you had fun!

Ceterum Censeo. . .



- Bug reports for E should include:
  - The exact command line leading to the bug
  - All input files needed to reproduce the bug
  - A description of what seems to be wrong
  - **The output of** `eprover --version`

References

References I

Leo Bachmair and Harald Ganzinger.

On Restrictions of Ordered Paramodulation with Simplification.
In M.E. Stickel, editor, *Proc. of the 10th CADE, Kaiserslautern*, volume 449 of *LNAI*, pages 427—441. Springer, 1990.

Leo Bachmair and Harald Ganzinger.

Rewrite-Based Equational Theorem Proving with Selection and Simplification.
Journal of Logic and Computation, 3(4):217–247, 1994.

Laura Kovács and Andrei Voronkov.

First-order theorem proving and Vampire.
In Natasha Sharygina and Helmut Veith, editors, *Proc. of the 25th CAV*, volume 8044 of *LNCS*, pages 1–35. Springer, 2013.

J. L. Lassez, Michael J. Maher, and Kim Marriott.

Unification revisited.
In Mauro Boscarol, Luigia Carlucci Aiello, and Giorgio Levi, editors, *Foundations of Logic and Functional Programming, Trento, Italy, December 15–19, 1986*, pages 67–113. Springer, 1988.

E.L. Lusk and R.A. Overbeek.

A Short Problem Set for Testing Systems that Include Equality Reasoning.
Technical report, Argonne National Laboratory, Illinois, 1982.

R. Nieuwenhuis and A. Rubio.

Paramodulation-Based Theorem Proving.
In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 7, pages 371–443. Elsevier Science and MIT Press, 2001.

References II

Robert Nieuwenhuis, José Miguel Rivero, and Miguel Ángel Vallejo.
Dedam: A Kernel of Data Structures and Algorithms for Automated Deduction with Equality Clauses.
In W.W. McCune, editor, *Proc. of the 14th CADE, Townsville*, volume 1249 of *LNAI*, pages 49–52. Springer, 1997.
Full version at `http://http://www.lsi.upc.es/~roberto/refs/cade1997.html`.

Stephan Schulz.
E – A Brainiac Theorem Prover.
Journal of AI Communications, 15(2/3):111–126, 2002.

Stephan Schulz.
Fingerprint Indexing for Paramodulation and Rewriting.
In Bernhard Gramlich, Ulrike Sattler, and Dale Miller, editors, *Proc. of the 6th IJCAR, Manchester*, volume 7364 of *LNAI*, pages 477–483. Springer, 2012.

Stephan Schulz.
Simple and Efficient Clause Subsumption with Feature Vector Indexing.
In Maria Paola Bonacina and Mark E. Stickel, editors, *Automated Reasoning and Mathematics: Essays in Memory of William W. McCune*, volume 7788 of *LNAI*, pages 45–67. Springer, 2013.

Stephan Schulz.
System Description: E 1.8.
In Ken McMillan, Aart Middeldorp, and Andrei Voronkov, editors, *Proc. of the 19th LPAR, Stellenbosch*, volume 8312 of *LNCS*, pages 735–743. Springer, 2013.

Stephan Schulz.
E 2.4 User Manual.
EasyChair preprint no. 2272, 2019.

References III

Stephan Schulz.

Lazy and eager patterns in high-performance automated theorem proving.
In Laura Kovács and Michael Rawson, editors, *Proceedings of the 7th and 8th Vampire Workshop*, volume 99 of *EPiC Series in Computing*, pages 7–12. EasyChair, 2024.

Stephan Schulz.

Shared terms and cached rewriting.
In Konstantin Korovin, Stephan Schulz, and Michael Rawson, editors, *Proceedings of the 14th and 15th International Workshops on the Implementation of Logics*, volume 21 of *Kalpa Publications in Computing*, pages 18–33. EasyChair, 2025.

Stephan Schulz and Martin Möhrmann.

Performance of clause selection heuristics for saturation-based theorem proving.
In Nicola Olivetti and Ashish Tiwari, editors, *Proc. of the 8th IJCAR, Coimbra*, volume 9706 of *LNAI*, pages 330–345. Springer, 2016.

R. Sekar, I.V. Ramakrishnan, and A. Voronkov.

Term Indexing.
In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 26, pages 1853–1961. Elsevier Science and MIT Press, 2001.

Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski.

SPASS Version 3.5.
In Renate Schmidt, editor, *Proc. of the 22nd CADE, Montreal, Canada*, volume 5663 of *LNAI*, pages 140–145. Springer, 2009.

Image Sources

- Public domain via the Wikimedia Commons
  - *Tour Eiffel* under construction
    `https://en.wikipedia.org/wiki/File:Construction_tour_eiffel4.JPG`
- Others: Painstakingly drawn by the author