Formal Methods in Software Engineering

Satisfiability Modulo Theories, Spring 2026

Konstantin Chukharev

Many-Sorted First-Order Logic

Why Many-Sorted?

In standard (mono-sorted) FOL, *all* variables range over a *single* domain. This is too restrictive for practical reasoning:

- **Programming languages** have *typed* variables: `int x`, `bool b`, `double[] arr`.
- **Databases** distinguish between strings, integers, dates, *etc.*
- **Mathematics** itself uses distinct sorts: $\mathbb{N}$, $\mathbb{R}$, matrices, functions, *etc.*

Key insight: *Many-sorted FOL* is a natural generalization of FOL where each variable, constant, and function is associated with a *sort* (type). This is exactly what SMT solvers use internally, and what SMT-LIB expresses.

Note: We have already seen first-order logic in the previous lecture. Here, we generalize it to the *many-sorted* setting, which is the standard framework for SMT.

From Single-Sorted to Many-Sorted FOL

In the previous lectures, we studied *single-sorted* (mono-sorted) FOL:

Single-sorted FOL (Weeks 4–5):

- One universal domain D
- All variables range over D
- $\forall x.\, P(x)$ means "for all x in D"
- Sufficient for pure math foundations

Many-sorted FOL (this lecture):

- Multiple domains: $\texttt{Int}^{\mathcal{I}}, \texttt{Bool}^{\mathcal{I}}, \texttt{Array}^{\mathcal{I}}, \ldots$
- Variables are *typed*: $x : \texttt{Int}$
- $\forall x : \sigma.\, P(x)$ — "for all x in $\sigma^{\mathcal{I}}$"
- Essential for programming & verification

The key extension: many-sorted FOL adds a *type system* to FOL. This is exactly what programming languages do — and why SMT solvers speak many-sorted logic natively.

Many-Sorted Signatures

Definition 1: A *many-sorted signature* $\Sigma = \langle \Sigma^S, \Sigma^F \rangle$ consists of:

- Σ^S — a set of *sorts* (also called *types*), *e.g.* `Bool`, `Int`, `Real`, `Array`.
- Σ^F — a set of *function symbols*, *e.g.* $=$, $+$, $<$, read, write.

Definition 2: Each *function symbol* $f \in \Sigma^F$ has a *rank* — an $(n + 1)$-tuple of sorts:

$$\operatorname{rank}(f) = \langle \sigma_1, ..., \sigma_n, \sigma_{n+1} \rangle$$

Intuitively, f takes n arguments of sorts $\sigma_1, ..., \sigma_n$ and returns a value of sort σ_{n+1}.

- Functions of arity 0 are called *constants*, with $\operatorname{rank}(f) = \langle \sigma \rangle$.
- Functions returning `Bool` are called *predicates*.

Note: Every signature includes a distinguished sort `Bool` with constants $\top$ and $\bot$, and an *equality predicate* $\doteq_\sigma$ with $\operatorname{rank}(\doteq_\sigma) = \langle \sigma, \sigma, \texttt{Bool} \rangle$ for each sort σ.

Signature Examples

Number Theory:

- $\Sigma^S = \{\mathtt{Nat}, \mathtt{Bool}\}$
- $\Sigma^F = \{0, S, <, +, \times, \doteq_{\mathtt{Nat}}, \ldots\}$
- $\mathrm{rank}(0) = \langle \mathtt{Nat} \rangle$
- $\mathrm{rank}(S) = \langle \mathtt{Nat}, \mathtt{Nat} \rangle$
- $\mathrm{rank}(<) = \langle \mathtt{Nat}, \mathtt{Nat}, \mathtt{Bool} \rangle$
- $\mathrm{rank}(+) = \langle \mathtt{Nat}, \mathtt{Nat}, \mathtt{Nat} \rangle$

Arrays:

- $\Sigma^S = \{\mathtt{Array}, \mathtt{I}, \mathtt{E}, \mathtt{Bool}\}$
- $\Sigma^F = \{\mathrm{read}, \mathrm{write}, \doteq_{\mathtt{Array}}, \ldots\}$
- $\mathrm{rank}(\mathrm{read}) = \langle \mathtt{Array}, \mathtt{I}, \mathtt{E} \rangle$
- $\mathrm{rank}(\mathrm{write}) = \langle \mathtt{Array}, \mathtt{I}, \mathtt{E}, \mathtt{Array} \rangle$

Example: *Propositional logic* can be viewed as a *one-sorted* theory:

- $\Sigma^S = \{\mathtt{Bool}\}$, $\Sigma^F = \{\neg, \wedge, \vee, p_1, p_2, \ldots\}$
- $\mathrm{rank}(p_i) = \langle \mathtt{Bool} \rangle$ (propositional variables are Boolean constants)
- $\mathrm{rank}(\neg) = \langle \mathtt{Bool}, \mathtt{Bool} \rangle$, $\mathrm{rank}(\wedge) = \langle \mathtt{Bool}, \mathtt{Bool}, \mathtt{Bool} \rangle$

Terms and Formulas

A *well-sorted term* of sort σ is built from variables, constants, and function applications that *respect* the rank:

- A variable x of sort σ is a term of sort σ.
- A constant c with $\mathrm{rank}(c) = \langle \sigma \rangle$ is a term of sort σ.
- If f has $\mathrm{rank}(f) = \langle \sigma_1, \ldots, \sigma_n, \sigma \rangle$ and $t_1, \ldots, t_n$ are terms of sorts $\sigma_1, \ldots, \sigma_n$, then $f(t_1, \ldots, t_n)$ is a term of sort σ.

Example: In the Number Theory signature with $x : \texttt{Nat}$:

- $S(0)$ is a well-sorted term of sort `Nat` ✓
- $S(x) + 0$ is a well-sorted term of sort `Nat` ✓
- $S(x < 0)$ is *not* well-sorted: $<$ returns `Bool`, but S expects `Nat` ✗

A *Σ-formula* is built from *atoms* (terms of sort `Bool`) using the standard logical connectives ($\neg$, $\wedge$, $\vee$, $\rightarrow$, $\Longleftrightarrow$) and quantifiers ($\forall x : \sigma.\, \varphi$, $\exists x : \sigma.\, \varphi$).

Many-Sorted Interpretations

Definition 3: A *many-sorted interpretation* $\mathcal{J}$ of a signature $\Sigma = \langle \Sigma^S, \Sigma^F \rangle$ assigns:

- To each sort $\sigma \in \Sigma^S$, a non-empty *domain* $\sigma^{\mathcal{J}}$ (*e.g.* $\texttt{Int}^{\mathcal{J}} = \mathbb{Z}$, $\texttt{Bool}^{\mathcal{J}} = \{\texttt{true}, \texttt{false}\}$).
- To each function symbol f with $\text{rank}(f) = \langle \sigma_1, \dots, \sigma_n, \sigma \rangle$, a function $f^{\mathcal{J}} : \sigma_1^{\mathcal{J}} \times \dots \times \sigma_n^{\mathcal{J}} \to \sigma^{\mathcal{J}}$.
- To each variable x of sort σ, a value $x^{\mathcal{J}} \in \sigma^{\mathcal{J}}$.

Example: For the Number Theory signature, one interpretation is:

- $\texttt{Nat}^{\mathcal{J}} = \mathbb{N}$, $0^{\mathcal{J}} = 0$, $S^{\mathcal{J}}(n) = n + 1$, $+^{\mathcal{J}}(m, n) = m + n$
- $\overset{\mathcal{J}}{<}(m, n) = \texttt{true}$ iff $m < n$ in the usual sense

This is the *intended* (standard) interpretation of natural numbers.

Mono-sorted vs. many-sorted: The only difference is that each variable, constant, and function is now tagged with a sort. Connective semantics ($\neg$, $\land$, $\lor$, $\forall$, $\exists$) remain exactly the same. Quantifiers now range over the domain of a *specific sort*: $(\forall x : \sigma.\, \varphi)$ means "for all x in $\sigma^{\mathcal{J}}$".

First-Order Theories

Motivation

Consider the signature $\Sigma = \langle \Sigma^S, \Sigma^F \rangle$ for a fragment of number theory:

- $\Sigma^S = \{\texttt{Nat}\}$, $\Sigma^F = \{0, 1, +, <\}$
- $\text{rank}(0) = \text{rank}(1) = \langle \texttt{Nat} \rangle$
- $\text{rank}(+) = \langle \texttt{Nat}, \texttt{Nat}, \texttt{Nat} \rangle$
- $\text{rank}(<) = \langle \texttt{Nat}, \texttt{Nat}, \texttt{Bool} \rangle$

1. Consider the Σ-sentence: $\forall x : \texttt{Nat}.\, \neg(x < x)$
   - Is it *valid*, that is, true under *all* interpretations?
   - No, *e.g.*, if we interpret $<$ as *equals* or *divides*.
2. Consider the Σ-sentence: $\neg \exists x : \texttt{Nat}.\, (x < 0)$
   - Is it *valid*?
   - No, *e.g.*, if we interpret $\texttt{Nat}$ as the set of *all* integers.
3. Consider the Σ-sentence: $\forall x : \texttt{Nat}. \forall y : \texttt{Nat}. \forall z : \texttt{Nat}.\, (x < y) \wedge (y < z) \rightarrow (x < z)$
   - Is it *valid*?
   - No, *e.g.*, if we interpret $<$ as the *successor* relation.

Motivation [2]

In practice, we often *do not care* about satisfiability or validity in *general*, but rather with respect to a *limited class* of interpretations.

A practical reason:

- When reasoning in a particular application domain, we typically have *specific* data types/structures in mind (*e.g.*, integers, strings, lists, arrays, finite sets, ...).
- More generally, we are typically *not* interested in *arbitrary* interpretations, but rather in *specific* ones.

Theories formalize this domain-specific reasoning: we talk about satisfiability and validity *with respect to a theory* or "*modulo a theory*".

A computational reason:

- The validity problem for FOL is *undecidable* in general.
- However, the validity problem for many *restricted* theories, is *decidable*.

First-Order Theories

Hereinafter, we assume that we have an infinite set of variables X.

Definition 4 (Theory): A first-order *theory* $\mathcal{T}$ is a pair[1] $\langle \Sigma, \boldsymbol{M} \rangle$, where

- $\Sigma = \langle \Sigma^S, \Sigma^F \rangle$ is a first-order signature,
- $\boldsymbol{M}$ is a class[2] of Σ-interpretations over X that is *closed under variable re-assignment*.

Definition 5: $\boldsymbol{M}$ is *closed under variable re-assignment* if every Σ-interpretation that differs from one in $\boldsymbol{M}$ in the way it interprets the variables in X is also in $\boldsymbol{M}$.

A theory limits the interpretations of Σ-formulas to those from $\boldsymbol{M}$.

[1]Here, we use **bold** style for $\boldsymbol{M}$ to denote that it is *not a single* model, but a *collection* of them.

[2]*Class* is a generalization of a set.

Theory Examples

Example: Theory of Real Arithmetic $\mathcal{T}_{\text{RA}} = \langle \Sigma_{\text{RA}}, \boldsymbol{M}_{\text{RA}} \rangle$:

- $\Sigma^S_{\text{RA}} = \{\texttt{Real}\}$
- $\Sigma^F_{\text{RA}} = \{+, -, \times, \leq\} \cup \{q \mid q \text{ is a decimal numeral}\}$
- All $\mathcal{I} \in \boldsymbol{M}_{\text{RA}}$ interpret `Real` as the set of *real numbers* $\mathbb{R}$, each q as the *decimal number* that it denotes, and the function symbols in the usual way.

Example: Theory of Ternary Strings $\mathcal{T}_{\text{TS}} = \langle \Sigma_{\text{TS}}, \boldsymbol{M}_{\text{TS}} \rangle$:

- $\Sigma^S_{\text{TS}} = \{\texttt{String}\}$
- $\Sigma^F_{\text{TS}} = \{\cdot\,, <\} \cup \{\text{a}, \text{b}, \text{c}\}$
- All $\mathcal{I} \in \boldsymbol{M}_{\text{TS}}$ interpret `String` as the set $\{\text{a}, \text{b}, \text{c}\}^*$ of all finite strings over the characters $\{\text{a}, \text{b}, \text{c}\}$, symbol $\cdot$ as string concatenation (*e.g.*, $\text{a} \cdot \text{b} = \text{ab}$), and $<$ as lexicographic order.

$\mathcal{T}$-interpretations

Definition 6 (Reduct): Let Σ and Ω be two signatures over variables X, where $\Omega \supseteq \Sigma$, that is, $\Omega^S \supseteq \Sigma^S$ and $\Omega^F \supseteq \Sigma^F$.

Let $\mathcal{I}$ be an Ω-interpretation over X.

The *reduct* $\mathcal{I}^\Sigma$ of $\mathcal{I}$ to Σ is a Σ-interpretation obtained from $\mathcal{I}$ by resticting it to the symbols in Σ.

Definition 7 ($\mathcal{T}$-interpretation): Given a theory $\mathcal{T} = \langle \Sigma, \boldsymbol{M} \rangle$, a *$\mathcal{T}$-interpretation* is any Ω-interpretation $\mathcal{I}$ for some signature $\Omega \supseteq \Sigma$ such that $\mathcal{I}^\Sigma \in \boldsymbol{M}$.

Note: This definition allows us to consider the satisfiability in a theory $\mathcal{T} = \langle \Sigma, \boldsymbol{M} \rangle$ of formulas that contain sorts or function symbols not in Σ. These symbols are usually called *uninterpreted* (in $\mathcal{T}$).

$\mathcal{T}$-interpretations [2]

Example: Consider again the theory of real arithmetic $\mathcal{T}_{\mathrm{RA}} = \langle \Sigma_{\mathrm{RA}}, \boldsymbol{M}_{\mathrm{RA}} \rangle$.

All $\mathcal{I} \in \boldsymbol{M}_{\mathrm{RA}}$ interpret `Real` as $\mathbb{R}$ and function symbols as usual.

Which of the following interpretations are $\mathcal{T}_{\mathrm{RA}}$-interpretations?

1. $\texttt{Real}^{\mathcal{I}_1} = \mathbb{Q}$, symbols in Σ^F_{RA} interpreted as usual. ✗
2. $\texttt{Real}^{\mathcal{I}_2} = \mathbb{R}$, symbols in Σ^F_{RA} interpreted as usual, and $\texttt{String}^{\mathcal{I}_2} = \{0.5, 1.3\}$. ✓
3. $\texttt{Real}^{\mathcal{I}_3} = \mathbb{R}$, symbols in Σ^F_{RA} interpreted as usual, and $\log^{\mathcal{I}_3}$ is the successor function. ✓

$\mathcal{T}$-satisfiability, $\mathcal{T}$-entailment, $\mathcal{T}$-validity

Definition 8 ($\mathcal{T}$-satisfiability): A Σ-formula α is *satisfiable in $\mathcal{T}$*, or *$\mathcal{T}$-satisfiable*, if it is satisfied by *some* $\mathcal{T}$-interpretation $\mathcal{I}$.

Definition 9 ($\mathcal{T}$-entailment): A set Γ of formulas *$\mathcal{T}$-entails* a formula α, if every $\mathcal{T}$-interpretation that satisfies all formulas in Γ also satisfies α.

Definition 10 ($\mathcal{T}$-validity): A formula α is *$\mathcal{T}$-valid*, if it is satisfied by *all* $\mathcal{T}$-interpretations.

Note: A formula α is *$\mathcal{T}$-valid* iff $\emptyset \vDash \alpha$.

Example: Which of the following Σ_{RA}-formulas is satisfiable or valid in $\mathcal{T}_{\mathrm{RA}}$?

1. $(x_0 + x_1 \leq 0.5) \land (x_0 - x_1 \leq 2)$ — *satisfiable*, *falsifiable*
2. $\forall x_0 .\, (x_0 + x_1 \leq 1.7) \to (x_1 \leq 1.7 - x_0)$ — *satisfiable*, *valid*
3. $\forall x_0 . \forall x_1 .\, (x_0 + x_1 \leq 1)$ — *unsatisfiable*, *falsifiable*

FOL vs Theory

For every signature Σ, entailment and validity in "pure" FOL can be seen as entailment and validity in the theory $\mathcal{T}_{\mathrm{FOL}} = \langle \Sigma, \boldsymbol{M}_{\mathrm{FOL}} \rangle$ where $\boldsymbol{M}_{\mathrm{FOL}}$ is the class of *all possible* Σ-interpretations.

- Pure first-order logic = reasoning over *all* possible interpretations.
- Reasoning modulo a theory = *restricting* interpretations with some domain constraints.
- Theories make automated reasoning *feasible* in many domains.

Axiomatization

Definition 11 (Axiomatic theory): A first-order *axiomatic theory* $\mathcal{T}$ is defined by a signature Σ and a set $\mathcal{A}$ of Σ-sentences, or *axioms*.

Definition 12 ($\mathcal{T}$-validity in axiomatic theory): An Ω-formula α is *valid* in an axiomatic theory $\mathcal{T}$ if it is entailed by the axioms of $\mathcal{T}$, that is, every Ω-interpretation $\mathcal{I}$ that satisfies $\mathcal{A}$ also satisfies α.

Note: Axiomatic theories are a *special case* of the general definition (via $\boldsymbol{M}$) of theories.

- Given an axiomatic theory $\mathcal{T}'$ defined by Σ and $\mathcal{A}$, we can define a theory $\mathcal{T} = \langle \Sigma, \boldsymbol{M} \rangle$ where $\boldsymbol{M}$ is the class of all Σ-interpretations that satisfy all axioms in $\mathcal{A}$.
- It is not hard to show that a formula α is valid in $\mathcal{T}$ *iff* it is valid in $\mathcal{T}'$.

Note: Not all theories are first-order axiomatizable.

Non-Axiomatizable Theories

Note: Not all theories are first-order axiomatizable.

Example: Consider the theory $\mathcal{T}_{\texttt{Nat}}$ of the natural numbers, with signature Σ with $\Sigma^S = \{\texttt{Nat}\}$, $\Sigma^F = \{0, S, +, <\}$, and $M = \{\mathcal{I}\}$ where $\texttt{Nat}^{\mathcal{I}} = \mathbb{N}$ and Σ^F is interpreted as usual.

Any set of axioms (for example, *Peano axioms*) for this theory is satisfied by *non-standard models*, *e.g.*, interpretations $\mathcal{I}'$ where $\texttt{Nat}^{\mathcal{I}'}$ includes other chains of elements besides the natural numbers.

However, these models *falsify* formulas that are *valid* in $\mathcal{T}_{\texttt{Nat}}$.

For example, "every number is either zero or a successor": $\forall x. \, (x \doteq 0) \lor \exists y. \, (x \doteq S(y))$.

- `true` in the *standard* model, *i.e.* $\texttt{Nat}^{\mathcal{I}} = \mathbb{N} = \{0, 1 := S(0), 2 := S(1), ...\}$.
- `false` in *non-standard* models, *e.g.*, $\texttt{Nat}^{\mathcal{I}'} = \{0, 1, 2, ...\} \cup \{\omega, \omega + 1, ...\}$
  - Intuitively, ω is "an infinite element".
  - The successor function still applies: $S(\omega) = \omega + 1$, $S(\omega + 1) = \omega + 2$, *etc.*
  - Even the addition and multiplication still works: $\omega + 3 = S(S(S(\omega)))$, $\omega \times 2 = \omega + \omega$.
  - But ω is larger than all standard numbers: $\omega > 0, \omega > 1, ...$

Peano Arithmetic

Definition 13: *Peano arithmetic* $\mathcal{T}_{\mathrm{PA}}$, or *first-order arithmetic*, is the axiomatic theory of natural numbers with signature $\Sigma_{\mathrm{PA}}^{F} = \{0, S, +, \times, =\}$ and *Peano axioms*:

1. $\forall x.\, (S(x) \neq 0)$ (zero)
2. $\forall x. \forall y.\, (S(x) = S(y)) \to (x = y)$ (successor)
3. $F[0] \land (\forall x.\, F[x] \to F[x + 1]) \to \forall x.\, F[x]$ (induction)
4. $\forall x.\, (x + 0 = x)$ (plus zero)
5. $\forall x. \forall y.\, (x + S(y) = S(x + y))$ (plus successor)
6. $\forall x.\, (x \times 0 = 0)$ (times zero)
7. $\forall x. \forall y.\, (x \times S(y) = (x \times y) + x)$ (times successor)

Axiom (induction) is the *induction axiom schema*. It stands for an *infinite* set of axioms, one for each Σ_{PA}-formula F with one free variable. The notation $F[\alpha]$ means that F contains α as a sub-formula.

The *intended interpretation* (*standard models*) of $\mathcal{T}_{\mathrm{PA}}$ have the domain $\mathbb{N}$ and the usual interpretations of the function symbols as $0_{\mathbb{N}}$, $S_{\mathbb{N}}$, $+_{\mathbb{N}}$, and $\times_{\mathbb{N}}$.

Presburger Arithmetic

Note: Satisfiability and validity in $\mathcal{T}_{\mathrm{PA}}$ is undecidable. Therefore, we need a more restricted theory of arithmetic that does not include multiplication.

Definition 14: *Presburger arithmetic* $\mathcal{T}_{\mathbb{N}}$ is the axiomatic theory of natural numbers with signature $\Sigma_{\mathbb{N}}^{F} = \{0, S, +, =\}$ and the *subset* of *Peano axioms*:

1. $\forall x.\, (S(x) \neq 0)$ (zero)
2. $\forall x. \forall y.\, (S(x) = S(y)) \rightarrow (x = y)$ (successor)
3. $F[0] \land (\forall x.\, F[x] \rightarrow F[x + 1]) \rightarrow \forall x.\, F[x]$ (induction)
4. $\forall x.\, (x + 0 = x)$ (plus zero)
5. $\forall x. \forall y.\, (x + S(y) = S(x + y))$ (plus successor)

Note: Presburger arithmetic is decidable.

Completeness of Theories

Definition 15: A Σ-theory $\mathcal{T}$ is *complete* if for every Σ-sentence α, either α or $\neg\alpha$ is valid in $\mathcal{T}$.

Note: In a complete Σ-theory, every Σ-sentence is either valid or unsatisfiable.

Example: Any theory $\mathcal{T} = \langle \Sigma, \boldsymbol{M} \rangle$ where all interpretations in $\boldsymbol{M}$ only differ in how they interpret the variables (*e.g.*, $\mathcal{T}_{\mathrm{RA}}$) is *complete*.

Example: The axiomatic (mono-sorted) theory of *monoids* with $\Sigma^F = \{\,\cdot\,, \varepsilon\}$ and axioms

$$\forall x. \forall y. \forall z. \,(x \cdot y) \cdot z \doteq x \cdot (y \cdot z) \qquad \forall x. \,(x \cdot \varepsilon \doteq x) \qquad \forall x. \,(\varepsilon \cdot x \doteq x)$$

is *incomplete*. For example, the sentence $\forall x. \forall y. \,(x \cdot y \doteq y \cdot x)$ is `true` in some monoids (*e.g.* the addition of integers *is* commutative) but `false` in others (*e.g.* the concatenation of strings *is not* commutative).

Completeness of Theories [2]

Example: The axiomatic (mono-sorted) theory of *dense linear orders without endpoints* with $\Sigma^F = \{\prec\}$ and the following axioms is *complete*.

$$\forall x.\forall y.(x \prec y) \to \exists z.\,((x \prec z) \land (z \prec y)) \qquad \text{(dense)}$$

$$\forall x.\forall y.\,((x \prec y) \lor (y \prec x) \lor (x \doteq y)) \qquad \text{(linear)}$$

$$\forall x.\,\neg(x \prec x) \quad \forall x.\forall y.\forall z.\,((x \prec y) \land (y \prec z) \to (x \prec z)) \qquad \text{(orders)}$$

$$\forall x.\exists y.\,(y \prec x) \quad \forall x.\exists y.\,(x \prec y) \qquad \text{(without endpoints)}$$

Decidability and Fragments

Recall that a set A is *decidable* if there exists a *terminating* procedure that, given an input element a, returns (after *finite* time) either “yes” if $a \in A$ or “no” if $a \notin A$.

Definition 16: A theory $\mathcal{T} = \langle \Sigma, M \rangle$ is *decidable* if the set of all *$\mathcal{T}$-valid* Σ-formulas is decidable.

Definition 17: A *fragment* of $\mathcal{T}$ is a *syntactically-restricted subset* of $\mathcal{T}$-valid Σ-formulas.

Example: The *quantifier-free* fragment of $\mathcal{T}$ is the set of all $\mathcal{T}$-valid Σ-formulas *without quantifiers*. The *linear* fragment of $\mathcal{T}_{\mathrm{RA}}$ is the set of all $\mathcal{T}$-valid Σ_{RA}-formulas *without multiplication* ($\times$).

Axiomatizability

Definition 18: A theory $\mathcal{T} = \langle \Sigma, M \rangle$ is *recursively axiomatizable* if M is the class of all interpretations satisfying a *decidable set* of first-order axioms $\mathcal{A}$.

Theorem 1 (Lemma): Every recursively axiomatizable theory $\mathcal{T}$ admits a procedure $E_{\mathcal{T}}$ that *enumerates* all $\mathcal{T}$-valid formulas.

Theorem 2: For every *complete* and *recursively axiomatizable* theory $\mathcal{T}$, validity in $\mathcal{T}$ is decidable.

Proof: Given a formula α, use $E_{\mathcal{T}}$ to enumerate all valid formulas. Since $\mathcal{T}$ is complete, either α or $\neg\alpha$ will eventually (after *finite* time) be produced by $E_{\mathcal{T}}$. $\square$

Introduction to SMT

Common Theories in SMT

Satisfiability Modulo Theories (SMT) traditionally focuses on theories with *decidable quantifier-free fragments*.

SMT is concerned with (un)satisfiability, but recall that a formula α is *$\mathcal{T}$-valid* iff $\neg\alpha$ is *$\mathcal{T}$-unsatisfiable*.

Checking the (un)satisfiability of quantifier-free formulas in main background theories *efficiently* has a large number of applications in:

- hardware and software verification
- model checking
- symbolic execution
- compiler validation
- type checking
- planning and scheduling
- software synthesis
- cyber-security
- verifiable machine learning
- analysis of biological systems

Further, we are going to study:

- A few of those *theories* and their *decision procedures*.
- *Proof systems* to reason *modulo theories* automatically.

From Quantifier-Free Formulas to Conjunctions of Literals

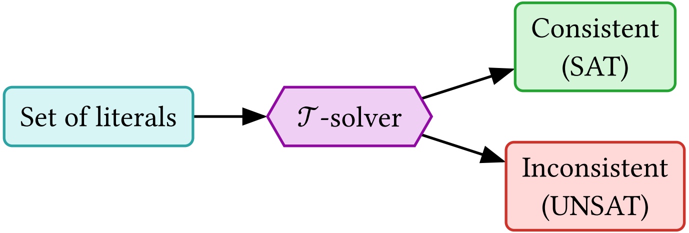
Theorem 3: The satisfiability of *quantifier-free* formulas in a theory $\mathcal{T}$ is *decidable* iff the satisfiability in $\mathcal{T}$ of *conjunctions of literals* is decidable.

Here, *literal* is an atom or its negation. For example: $(a \doteq b)$, $\neg(a + 1 < b)$, $(f(b) \doteq g(f(a)))$.

Proof: A quantifier-free formula can be transformed into disjunctive normal form (DNF), and its satisfiability reduces to checking satisfiability of conjunctions of literals. Conversely, a conjunction of literals is a special case of a quantifier-free formula. Thus, the two satisfiability problems are equivalent. $\square$

Theory Solvers

Definition 19 ($\mathcal{T}$-solver): A *theory solver*, or *$\mathcal{T}$-solver*, is a specialized decision procedure for the satisfiability of conjunctions of literals in a theory $\mathcal{T}$.



Key insight: A $\mathcal{T}$-solver is the bridge between SAT and theory reasoning. It answers: "Is this set of theory literals consistent in this domain?"

Theory of Uninterpreted Functions

Definition 20: Given a signature Σ, the most general theory consists of the class of *all* Σ-interpretations. In fact, this is a *family* of theories parameterized by the signature Σ.

It is known as the theory of *equality with uninterpreted functions* $\mathcal{T}_{\text{EUF}}$, or the *empty theory*, since it contains no *sentences*.

Example: $(a \doteq b) \land (f(a) \doteq b) \land \neg(g(a) \doteq g(f(a)))$ Is this formula satisfiable in $\mathcal{T}_{\text{EUF}}$?

Both validity and satisfiability are undecidable in $\mathcal{T}_{\text{EUF}}$.

- Validity in $\mathcal{T}_{\text{EUF}}$ is *semi-decidable* — this is just a validity in FOL.
- Since a formula α is $\mathcal{T}$-satisfiable iff $\neg\alpha$ is not $\mathcal{T}$-valid, $\mathcal{T}_{\text{EUF}}$-satisfiability is *co-recognizable*.

However, the satisfiability of *conjunctions of $\mathcal{T}_{\text{EUF}}$-literals* is *decidable*, in polynomial time, using the *congruence closure* algorithm.

Theory of Real Arithmetic

Definition 21: The theory of *real arithmetic* $\mathcal{T}_{\mathrm{RA}}$ is a theory of inequalities over the real numbers.

- $\Sigma^S = \{\texttt{Real}\}$
- $\Sigma^F = \{+, -, \times, <\} \cup \{q \mid q \text{ is a decimal numeral}\}$
- $\boldsymbol{M}$ is the class of interpretations that interpret `Real` as the set of *real numbers* $\mathbb{R}$, and the function symbols in the usual way.

Satisfiability in the full $\mathcal{T}_{\mathrm{RA}}$ is *decidable* (in worst-case doubly-exponential time).

Restricted fragments of $\mathcal{T}_{\mathrm{RA}}$ can be decided more efficiently.

Example: Quantifier-free linear real arithmetic (`QF_LRA`) is the theory of *linear* inequalities over the reals, where $\times$ can only be used in the form of *multiplication by constants* (decimal numerals).

The satisfiability of conjunctions of literals in `QF_LRA` is *decidable* in *polynomial time*.

Theory of Integer Arithmetic

Definition 22: The theory of *integer arithmetic* $\mathcal{T}_{\text{IA}}$ is a theory of inequalities over the integers.

- $\Sigma^S = \{\texttt{Int}\}$
- $\Sigma^F = \{+, -, \times, <\} \cup \{n \mid n \text{ is an integer numeral}\}$
- $\boldsymbol{M}$ is the class of interpretations that interpret `Int` as the set of *integers* $\mathbb{Z}$, and the function symbols in the usual way.

Satisfiability in $\mathcal{T}_{\text{IA}}$ is *not even semi-decidable*!

Satisfiability of quantifier-free Σ-formulas in $\mathcal{T}_{\text{IA}}$ is *undecidable* as well.

Linear integer arithmetic (`LIA`, also known as *Presburger arithmetic*) is decidable, but not efficiently (in worst-case triply-exponential time). Its quantifier-free fragment (`QF_LIA`) is NP-complete.

Theory of Arrays with Extensionality

Definition 23: The theory of *arrays* $\mathcal{T}_{\mathrm{AX}}$ is useful for modelling RAM or array data structures.

- $\Sigma^S = \{\mathtt{A}, \mathtt{I}, \mathtt{E}\}$ (arrays, indices, elements)
- $\Sigma^F = \{\mathrm{read}, \mathrm{write}\}$, where $\mathrm{rank}(\mathrm{read}) = \langle \mathtt{A}, \mathtt{I}, \mathtt{E} \rangle$ and $\mathrm{rank}(\mathrm{write}) = \langle \mathtt{A}, \mathtt{I}, \mathtt{E}, \mathtt{A} \rangle$

Let a be a variable of sort $\mathtt{A}$, variable i of sort $\mathtt{I}$, and variable v of sort $\mathtt{E}$.

- $\mathrm{read}(a, i)$ denotes the value stored in array a at index i.
- $\mathrm{write}(a, i, v)$ denotes the array that stores value v at index i and is otherwise identical to a.

Example: $\mathrm{read}(\mathrm{write}(a, i, v), i) \doteq_{\mathtt{E}} v$

- Is this formula *intuitively* valid/satisfiable/unsatisfiable in $\mathcal{T}_{\mathrm{A}}$?

Example: $\forall i.\, (\mathrm{read}(a, i) \doteq_{\mathtt{E}} \mathrm{read}(a', i)) \to (a \doteq_{\mathtt{A}} a')$

- Is this formula *intuitively* valid/satisfiable/unsatisfiable in $\mathcal{T}_{\mathrm{A}}$?

Theory of Arrays with Extensionality [2]

Definition 24: The theory of arrays $\mathcal{T}_{\text{AX}} = \langle \Sigma, M \rangle$ is finitely axiomatizable.

M is the class of interpretations that satisfy the following axioms:

1. $\forall a. \forall i. \forall v. \, (\text{read}(\text{write}(a, i, v), i) \doteq_{\mathsf{E}} v)$
2. $\forall a. \forall i. \forall j. \forall v. \, \neg(i \doteq_{\mathsf{I}} j) \to (\text{read}(\text{write}(a, i, v), j) \doteq_{\mathsf{E}} \text{read}(a, j))$
3. $\forall a. \forall b. \, (\forall i. \, (\text{read}(a, i) \doteq_{\mathsf{E}} \text{read}(b, i))) \to (a \doteq_{\mathsf{A}} b)$

Note: The last axiom is called *extensionality* axiom. It states that two arrays are equal if they have the same values at all indices. It can be omitted to obtain a theory of arrays *without extensionality* $\mathcal{T}_{\text{A}}$.

Validity and satisfiability in $\mathcal{T}_{\text{AX}}$ is *undecidable*.

There are several *decidable fragments* of $\mathcal{T}_{\text{A}}$.

Survey of Decidability and Complexity

Theory	Description	Full	QF	Full complexity	QFC complexity
PL	Propositional Logic	—	yes	NP-complete	$\Theta(n)$
$\mathcal{T}_{\mathrm{EUF}}$	Equality	no	yes	undecidable	$\mathcal{O}(n \log n)$
$\mathcal{T}_{\mathrm{PA}}$	Peano Arithmetic	no	no	undecidable	undecidable
$\mathcal{T}_{\mathbb{N}}$	Presburger Arithmetic	yes	yes	$\Omega(2^{2^n}), \mathcal{O}\left(2^{2^{2^{kn}}}\right)$	NP-complete
$\mathcal{T}_{\mathbb{Z}}$	Linear Integers (LIA)	yes	yes	$\Omega(2^{2^n}), \mathcal{O}\left(2^{2^{2^{kn}}}\right)$	NP-complete
$\mathcal{T}_{\mathbb{R}}$	Reals	yes	yes	$\mathcal{O}\left(2^{2^{kn}}\right)$	$\mathcal{O}\left(2^{2^{kn}}\right)$
$\mathcal{T}_{\mathbb{Q}}$	Linear Rationals	yes	yes	$\Omega(2^n), \mathcal{O}\left(2^{2^{kn}}\right)$	PTIME
$\mathcal{T}_{\mathrm{RDS}}$	Recursive Data Structures	no	yes	undecidable	$\mathcal{O}(n \log n)$
$\mathcal{T}_{\mathrm{ARDS}}$	Acyclic RDS	yes	yes	not elementary recursive	$\Theta(n)$
$\mathcal{T}_{\mathrm{A}}$	Arrays	no	yes	undecidable	NP-complete
$\mathcal{T}_{\mathrm{AX}}$	Arrays with Extensionality	no	yes	undecidable	NP-complete

Survey of Decidability and Complexity [2]

Legend for the table:

- **"Full"** denotes the decidability of a complete theory *with* quantifiers.
- **"QF"** denotes the decidability of a *quantifier-free* theory.
- **"Full complexity"** denotes the complexity of the satisfiability in a complete theory *with quantifiers*.
- **"QFC complexity"** denotes the complexity of the satisfiability in a *quantifier-free conjunctive* fragment.
- For complexities, n is the size of the input formula, k is some positive integer.
- "*Not elementary recursive*" means the runtime cannot be bounded by a fixed-height stack of exponentials.

Roadmap

In the following sections, we will study *theory solvers* for several important theories:

1. **Difference Logic** (DL) — a very restricted fragment of integer arithmetic; solved via *graph-based* cycle detection.
2. **Equality with Uninterpreted Functions** ($\mathcal{T}_{\mathrm{EUF}}$) — the “empty” theory; solved via *congruence closure*.
3. **Arrays** ($\mathcal{T}_{\mathrm{AX}}$) — models memory/arrays; extends the EUF proof system.
4. **Linear Real Arithmetic** ($\mathcal{T}_{\mathrm{RA}}$) — linear inequalities over $\mathbb{R}$; solved via the *Simplex* algorithm.

We will then see how to *combine* theory solvers (Nelson-Oppen method), build a complete *SMT solver* (CDCL($\mathcal{T}$) architecture), and use the *SMT-LIB* standard language with the Z3 solver.

Difference Logic

Difference Logic

Definition 25: *Difference logic* (DL) is a fragment of linear integer arithmetic consisting of conjunctions of literals of the very restricted form:

$$x - y \bowtie c$$

where x and y are integer variables, c is a numeral, and $\bowtie \in \{=, <, \leq, >, \geq\}$.

A solver for difference logic consists of three steps:

1. Literals normalization.
2. Conversion to a graph.
3. Cycle detection.

Decision Procedure for DL

Step 1: Rewrite each literal using $\leq$ by applying the following rules:

1. $(x - y = c) \rightarrow (x - y \leq c) \wedge (x - y \geq c)$
2. $(x - y \geq c) \rightarrow (y - x \leq -c)$
3. $(x - y > c) \rightarrow (y - x < -c)$
4. $(x - y < c) \rightarrow (x - y \leq c - 1)$

Step 2: Construct a weighted directed graph G with a vertex for each variable and an edge $x \xrightarrow{c} y$ for each literal $(x - y \leq c)$.

Step 3: Check for *negative cycles* in G.

- Use, for example, the Bellman-Ford algorithm.
- If G contains a negative cycle, the set of literals is *inconsistent* (UNSAT).
- Otherwise, the set of literals is *consistent* (SAT).

Difference Logic Example

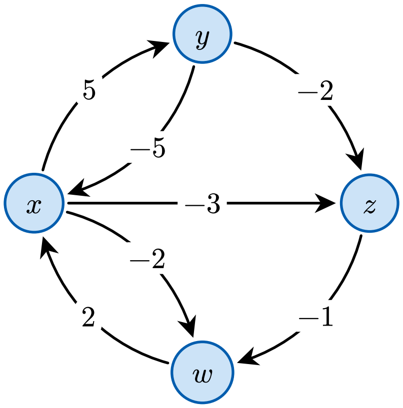
Consider the following set of difference logic literals:

$$(x - y = 5) \land (z - y \geq 2) \land (z - x > 2) \land (w - x = 2) \land (z - w < 0)$$

Normalize the literals:

- $(x - y = 5) \implies (x - y \leq 5) \land (y - x \leq -5)$
- $(z - y \geq 2) \implies (y - z \leq -2)$
- $(z - x > 2) \implies (x - z \leq -3)$
- $(w - x = 2) \implies (w - x \leq 2) \land (x - w \leq -2)$
- $(z - w < 0) \implies (z - w \leq -1)$

UNSAT because of the negative cycle: $x \xrightarrow{-3} z \xrightarrow{-1} w \xrightarrow{2} x$.



Equality

Theory of Equality with Uninterpreted Functions

Definition 26: The theory of equality with uninterpreted functions $\mathcal{T}_{\text{EUF}}$ is defined by the signature $\Sigma^F = \{\doteq, f, g, h, \ldots\}$ (*interpreted* equality and *uninterpreted* functions) and the following axioms:

1. $\forall x.\, x \doteq x$ (reflexivity)
2. $\forall x. \forall y.\, (x \doteq y) \to (y \doteq x)$ (symmetry)
3. $\forall x. \forall y. \forall z.\, (x \doteq y) \land (y \doteq z) \to (x \doteq z)$ (transitivity)
4. $\forall \boldsymbol{x}. \forall \boldsymbol{y}.\, (\boldsymbol{x} = \boldsymbol{y}) \to (f(\boldsymbol{x}) \doteq f(\boldsymbol{y}))$ (function congruence)

Flattening

Definition 27: A literal is *flat* if it is of the form:

- $x \doteq y$
- $\neg(x \doteq y)$
- $x \doteq f(\boldsymbol{z})$

where x and y are variables, f is a function symbol, and $\boldsymbol{z}$ is a tuple of 0 or more variables.

Note: Any set of literals can be converted to an equisatisfiable set of *flat* literals by introducing *new* variables and equating non-equational atoms to `true`.

Example: Consider the set of literals: $\{x + y > 0, y \doteq f(g(z))\}$.

We can convert it to an equisatisfiable set of flat literals by introducing fresh variables v_i:

$$\{\, v_1 \doteq v_2 > v_3, \quad v_1 \doteq \texttt{true}, \quad v_2 \doteq x + y, \quad v_3 \doteq 0, \quad y \doteq f(v_4), \quad v_4 \doteq g(z) \,\}$$

Hereinafter, we will assume that all literals are *flat*.

Notation and Assumptions

- We abbreviate $\neg(s \doteq t)$ with $s \not\doteq t$.
- For tuples $\boldsymbol{u} = \langle u_1, \ldots, u_n \rangle$ and $\boldsymbol{v} = \langle v_1, \ldots, v_n \rangle$, we abbreviate $(u_1 \doteq v_1) \wedge \ldots \wedge (u_n \doteq v_n)$ with $\boldsymbol{u} = \boldsymbol{v}$.
- Γ is used to refer to the "current" proof state in rule premises.
- $\Gamma, s \doteq t$ is an abbreviation for $\Gamma \cup \{s \doteq t\}$.
- If applying a rule R does not change Γ, then R *is not applicable* to Γ, that is, Γ is *irreducible* w.r.t. R.

Satisfiability Proof System for `QF_UF`

Let `QF_UF` be the quantifier-free fragment of FOL over some signature Σ.

Below is a simple *satisfiability proof system* R_{UF} for `QF_UF`:

$$\textsc{Refl}\ \frac{x \text{ occurs in } \Gamma}{\Gamma := \Gamma, x \doteq x} \qquad \textsc{Cong}\ \frac{x \doteq f(\boldsymbol{u}) \in \Gamma \quad y \doteq f(\boldsymbol{v}) \in \Gamma \quad \boldsymbol{u} = \boldsymbol{v} \in \Gamma}{\Gamma := \Gamma, x \doteq y}$$

$$\textsc{Symm}\ \frac{x \not\doteq y \in \Gamma}{\Gamma := \Gamma, y \doteq x} \qquad \textsc{Contr}\ \frac{x \doteq y \in \Gamma \quad x \not\doteq y \in \Gamma}{\text{UNSAT}}$$

$$\textsc{Trans}\ \frac{x \not\doteq y \in \Gamma \quad y \doteq z \in \Gamma}{\Gamma := \Gamma, x \doteq z} \qquad \textbf{SAT}\ \frac{\text{No other rules apply}}{\text{SAT}}$$

Is R_{UF} *sound*?

Is R_{UF} *terminating*?

Example Derivation in R_{UF}

Example: Determine the satisfiability of the following set of literals: $a \doteq f(f(a))$, $a \doteq f(f(f(a)))$, $g(a, f(a)) \not\doteq g(f(a), a)$. Flatten the literals and construct the following proof:

$$
\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{a \doteq f(a_1), a_1 \doteq f(a), a \doteq f(a_2), a_2 \doteq f(a_1), a_3 \not\doteq a_4, a_3 \doteq g(a, a_1), a_4 \doteq g(a_1, a)}{a_1 \doteq a_1}\ \textsc{Refl}}{a \doteq a_2}\ \textsc{Cong}\text{ applied to } a \doteq f(a_1), a_2 \doteq f(a_1), a_1 \doteq a_1}{a_1 \doteq a}\ \textsc{Cong}\text{ applied to } a_1 \doteq f(a), a \doteq f(a_2), a \doteq a_2}{a \doteq a_1}\ \textsc{Symm}}{a_3 \doteq a_4}\ \textsc{Cong}\text{ applied to } a_3 \doteq g(a, a_1), a_4 \doteq g(a_1, a), a \doteq a_1, a_1 \doteq a}{\text{UNSAT}}\ \textsc{Contr}\text{ applied to } a_3 \doteq a_4, a_3 \not\doteq a_4
$$

Soundness of R_{UF}

Theorem 4 (Refutation soundness): A literal set Γ_0 is unsatisfiable if R_{UF} derives UNSAT from it.

Proof: All rules except SAT are satisfiability-preserving.

If a derivation from Γ_0 ends with UNSAT, then Γ_0 must be unsatisfiable. □

Theorem 5 (Solution soundness): A literal set Γ_0 is satisfiable if R_{UF} derives SAT from it.

Proof: Let Γ be a proof state to which SAT applies. From Γ, we can construct an interpretation $\mathcal{J}$ that satisfies Γ_0. Let $s \sim t$ iff $(s \doteq t) \in \Gamma$. One can show that $\sim$ is an equivalence relation.

Let the domain of $\mathcal{J}$ be the equivalence classes $E_1, \ldots, E_k$ of $\sim$.

- For every variable or a constant t, let $t^{\mathcal{J}} = E_i$ if $t \in E_i$ for some i. Otherwise, let $t^{\mathcal{J}} = E_1$.
- For every unary function symbol f, and equivalence class E_i, let $f^{\mathcal{J}}$ be such that $f^{\mathcal{J}}(E_i) = E_j$ if $f(t) \in E_j$ for some $t \in E_i$. Otherwise, let $f^{\mathcal{J}}(E_i) = E_1$. Define $f^{\mathcal{J}}$ for non-unary f similarly.

We can show that $\mathcal{J} \vDash \Gamma$. This means that $\mathcal{J}$ models Γ_0 as well since $\Gamma_0 \subseteq \Gamma$. □

Termination in R_{UF}

Theorem 6: Every derivation strategy for R_{UF} terminates.

Proof: R_{UF} adds to the current state Γ only equalities between variables of Γ_0.

So, at some point it will run out of new equalities to add. $\square$

Completeness of R_{UF}

Theorem 7 (Refutation completeness): Every derivation strategy applied to an unsatisfiable state Γ_0 ends with UNSAT.

Proof: Let Γ_0 be an unsatisfiable state. Suppose there was a derivation from Γ_0 that did not end with UNSAT. Then, by the termination theorem, it would have to end with SAT. But then R_{UF} would be not be solution sound. □

Theorem 8 (Solution completeness): Every derivation strategy applied to a satisfiable state Γ_0 ends with SAT.

Proof: Let Γ_0 be a satisfiable state. Suppose there was a derivation from Γ_0 that did not end with SAT. Then, by the termination theorem, it would have to end with UNSAT. But then R_{UF} would be not be refutation sound. □

Arrays

Theory of Arrays

Definition 28: The theory of *arrays* $\mathcal{T}_{\mathrm{AX}}$ is defined by the signature $\Sigma^S = \{\mathtt{A}, \mathtt{I}, \mathtt{E}\}$ (arrays, indices, elements), $\Sigma^F = \{\mathrm{read}, \mathrm{write}\}$ and the following axioms:

1. $\forall a. \forall i. \forall v. \, (\mathrm{read}(\mathrm{write}(a, i, v), i) \doteq_{\mathtt{E}} v)$
2. $\forall a. \forall i. \forall j. \forall v. \, \neg(i \doteq_{\mathtt{I}} j) \to (\mathrm{read}(\mathrm{write}(a, i, v), j) \doteq_{\mathtt{E}} \mathrm{read}(a, j))$
3. $\forall a. \forall b. \, (\forall i. \, (\mathrm{read}(a, i) \doteq_{\mathtt{E}} \mathrm{read}(b, i))) \to (a \doteq_{\mathtt{A}} b)$

Example

```
void ReadBlock(int data[], int x, int len) {
  int i = 0;
  int next = data[0];
  for (; i < next && i < len; i = i + 1) {
    if (data[i] == x)
      break;
    else
      Process(data[i]);
  }
  assert(i < len);
}
```

One pass through this code can be translated into the following $\mathcal{T}_{\mathrm{A}}$ formula:

$$(\mathrm{i} \doteq 0) \wedge (\mathrm{next} \doteq \mathrm{read}(\mathrm{data}, 0)) \wedge (\mathrm{i} < \mathrm{next}) \wedge$$
$$\wedge\ (\mathrm{i} < \mathrm{len}) \wedge (\mathrm{read}(\mathrm{data}, \mathrm{i}) \doteq x) \wedge \neg(\mathrm{i} < \mathrm{len})$$

Satisfiability Proof System for `QF_AX`

The satisfiability proof system R_{AX} for $\mathcal{T}_{\text{AX}}$ *extends* the proof system R_{UF} for $\mathcal{T}_{\text{UF}}$ with the following rules:

$$\textbf{RIntro1}\ \frac{b \doteq \text{write}(a, i, v) \in \Gamma}{\Gamma := \Gamma, v \doteq \text{read}(b, i)}$$

$$\textbf{RIntro2}\ \frac{b \doteq \text{write}(a, i, v) \in \Gamma \qquad u \doteq \text{read}(x, j) \in \Gamma \qquad x \in \{a, b\}}{\Gamma := \Gamma, i \doteq j \qquad \Gamma := \Gamma, i \not\doteq j, u \doteq \text{read}(a, j), u \doteq \text{read}(b, j)}$$

$$\textbf{Ext}\ \frac{a \not\doteq b \in \Gamma \qquad a \text{ and } b \text{ are arrays}}{\Gamma := \Gamma, u \not\doteq v, u \doteq \text{read}(a, k), v \doteq \text{read}(b, k)}$$

- **RIntro1**: After writing v at index i, the reading at the same index i gives us back the value v.
- **RIntro2**: After writing v in a at index i, the reading from a or b at index j *splits* in two cases: (1) i equals j, (2) a and b have the same value u at position j.
- **Ext**: If two arrays a and b are distinct, they must differ at some index k.

Example Derivation in R_{AX}

Example: Determine the satisfiability of $\{\mathrm{write}(a_1, i, \mathrm{read}(a_1, i)) \doteq \mathrm{write}(a_2, i, \mathrm{read}(a_2, i)), a_1 \not\doteq a_2\}$.

First, flatten the literals:

$$\{\mathrm{write}(a_1, i, \mathrm{read}(a_1, i)) \doteq \mathrm{write}(a_2, i, \mathrm{read}(a_2, i))\} \rightarrow$$
$$\rightarrow \{a_1' \doteq a_2', a_1' \doteq \mathrm{write}(a_1, i, \mathrm{read}(a_2, i)), a_2' \doteq \mathrm{write}(a_2, i, \mathrm{read}(a_1, i)), a_1 \not\doteq a_2\} \rightarrow$$
$$\rightarrow \{a_1' \doteq a_2', a_1' \doteq \mathrm{write}(a_1, i, v_2), v_2 \doteq \mathrm{read}(a_2, i), a_2' \doteq \mathrm{write}(a_2, i, v_1), v_1 \doteq \mathrm{read}(a_1, i), a_1 \not\doteq a_2\}$$

Example Derivation in R_{AX} [2]

1. $a'_1 \doteq a'_2, a'_1 \doteq \mathrm{write}(a_1, i, v_2), v_2 \doteq \mathrm{read}(a_2, i), a'_2 \doteq \mathrm{write}(a_2, i, v_1), v_1 \doteq \mathrm{read}(a_1, i), a_1 \not\doteq a_2$
2. (by Refl) $a_1 \doteq a_1$
3. (by Refl) $a_2 \doteq a_2$
4. (by Ext) $u_1 \not\doteq u_2, u_1 \doteq \mathrm{read}(a_1, n), u_2 \doteq \mathrm{read}(a_2, n)$
5. (by RIntro2) split

---

6. $i \doteq n$
7. (by Cong) $v_1 \doteq u_1$
8. (by Symm) $u_1 \doteq v_1$
9. (by Cong) $v_2 \doteq u_2$
10. (by RIntro1) $v_2 \doteq \mathrm{read}(a'_1, i)$
11. (by RIntro1) $v_1 \doteq \mathrm{read}(a'_2, i)$
12. (by Refl) $i \doteq i$
13. (by Cong) $v_1 \doteq v_2$
14. (by Trans) $u_1 \doteq u_2$
15. (by Contr) UNSAT

---

6. $i \not\doteq n, u_1 \doteq \mathrm{read}(a'_1, n)$
7. (by RIntro2) split

---

8. $i \doteq n$
9. (by Contr) UNSAT

---

8. $i \not\doteq n, u_2 \doteq \mathrm{read}(a'_2, n)$
9. (by Refl) $n \doteq n$
10. (by Cong) $u_1 \doteq u_2$
11. (by Contr) UNSAT

Arithmetic

Theory of Real Arithmetic

Definition 29: The theory of *real arithmetic* $\mathcal{T}_{\mathrm{RA}}$ is defined by the signature $\Sigma^S_{\mathrm{RA}} = \{\texttt{Real}\}$, $\Sigma^F_{\mathrm{RA}} = \{+, -, \times, \leq\} \cup \{q \mid q \text{ is a decimal numeral}\}$ and the class of interpretations $\boldsymbol{M}_{\mathrm{RA}}$ that interpret `Real` as the set of *real numbers* $\mathbb{R}$, and the function symbols in the usual way.

Quantifier-free linear real arithmetic (`QF_LRA`) is the theory of *linear inequalities* over the reals, where $\times$ can only be used in the form of *multiplication by constants (decimal numerals)*.

Linear Programming

Definition 30: A *linear program* (LP) consists of:

1. An $m \times n$ matrix $\boldsymbol{A}$, the *contraint matrix*.
2. An m-dimensional vector $\boldsymbol{b}$.
3. An n-dimensional vector $\boldsymbol{c}$, the *objective function*.

Let $\boldsymbol{x}$ be a vector of n variables.

Goal: Find a solution $\boldsymbol{x}$ that *maximizes* $\boldsymbol{c}^T\boldsymbol{x}$ subject to the linear constraints $\boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}$ (and[3] $\boldsymbol{x} \geq \boldsymbol{0}$).

Note: All **bold**-styled symbols denote *vectors* or *matrices*, *e.g.*, $\boldsymbol{x}$, $\boldsymbol{A}$, $\boldsymbol{0}$.

[3] The constraint $\boldsymbol{x} \geq \boldsymbol{0}$ is introduced when LP is expressed in *standard form*, explained later in these slides.

Example and Terminology

Example: Maximize $2x_2 - x_1$ subject to:

$$x_1 + x_2 \leq 3$$
$$2x_1 - x_2 \leq -5$$

Here, $\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$, $\boldsymbol{A} = \begin{bmatrix} 1 & 1 \\ 2 & -1 \end{bmatrix}$, $\boldsymbol{b} = \begin{bmatrix} 3 \\ -5 \end{bmatrix}$, $\boldsymbol{c} = \begin{bmatrix} -1 \\ 2 \end{bmatrix}$.

Find $\boldsymbol{x}$ that maximizes $\boldsymbol{c}^T \boldsymbol{x}$ subject to $\boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}$.

Definition 31: An assignment of $\boldsymbol{x}$ is a *feasible solution* if it satisfies $\boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}$.

- Is $\boldsymbol{x} = \langle 0, 0 \rangle$ a feasible solution? ✗
- Is $\boldsymbol{x} = \langle -2, 1 \rangle$ a feasible solution? ✓

Definition 32: For a given assignment $\boldsymbol{x}$, the value $\boldsymbol{c}^T \boldsymbol{x}$ is the *objective value*, or *cost*, of $\boldsymbol{x}$.

- What is the objective value of $\boldsymbol{x} = \langle -2, 1 \rangle$?

Example and Terminology [2]

Definition 33: An *optimal solution* is a feasible solution with a *maximal* objective value among all feasible solutions.

Definition 34: If a linear program has no feasible solutions, it is *infeasible*.

Definition 35: The linear program is *unbounded* if the objective value of the optimal solution is ∞.

Geometric Interpretation

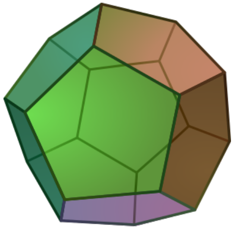
Definition 36: A *polytope* is a generalization of 3-dimensional polyhedra to higher dimensions.

Definition 37: A polytope P is *convex* if every point on the line segment connecting any two points in P is also within P.

Formally, for all $a, b \in \mathbb{R}^n \cap P$, and for all $\lambda \in [0; 1]$, it holds that $\lambda a + (1 - \lambda) b \in P$.

Note: For an $m \times n$ constraint matrix $\boldsymbol{A}$, the set of points $P = \{\boldsymbol{x} \mid \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}\}$ forms a *convex polytope* in n-dimensional space.

LP goal: find a point $\boldsymbol{x}$ *inside the polytope* that maximizes $\boldsymbol{c}^T \boldsymbol{x}$ for a given $\boldsymbol{c}$.



Note: LP is *infeasible* iff the polytope is *empty*.

Note: LP is *unbounded* iff the polytope is *open* in the direction of the objective function.

Note: The *optimal solution* for a bounded LP lies on a *vertex* of the polytope.

Satisfiability as Linear Programming

Our goal is to use LP to check the satisfiability of *sets of linear $\mathcal{T}_{\mathrm{RA}}$-literals*.

Step 1: Convert equalities to inequalities.

- A linear $\mathcal{T}_{\mathrm{RA}}$-equality can be written to have the form $\boldsymbol{a}^T\boldsymbol{x} = \boldsymbol{b}$.
- We rewrite this further as $\boldsymbol{a}^T\boldsymbol{x} \geq \boldsymbol{b}$ and $\boldsymbol{a}^T\boldsymbol{x} \leq \boldsymbol{b}$.
- And finally to $-\boldsymbol{a}^T\boldsymbol{x} \leq -\boldsymbol{b}$ and $\boldsymbol{a}^T\boldsymbol{x} \leq \boldsymbol{b}$.

Step 2: Handle inequalities.

- A $\mathcal{T}_{\mathrm{RA}}$-literal of the form $\boldsymbol{a}^T\boldsymbol{x} \leq \boldsymbol{b}$ is already in the desired form.
- A $\mathcal{T}_{\mathrm{RA}}$-literal of the form $\neg(\boldsymbol{a}^T\boldsymbol{x} \leq \boldsymbol{b})$ is transformed as follows:

  $$\neg(\boldsymbol{a}^T\boldsymbol{x} \leq \boldsymbol{b}) \to (\boldsymbol{a}^T\boldsymbol{x} > \boldsymbol{b}) \to (-\boldsymbol{a}^T\boldsymbol{x} < -\boldsymbol{b}) \to (-\boldsymbol{a}^T\boldsymbol{x} + y \leq -\boldsymbol{b}), (y > 0)$$

  where y is a fresh variable used for all negated inequalities.

  Example: $\neg(2x_1 - x_2 \leq 3)$ rewrites to $-2x_1 + x_2 + y \leq -3$, $y > 0$
- If there are no negated inequalities, add the inequality $y \leq 1$, where y is a fresh variable.

Satisfiability as Linear Programming [2]

- In either case, we end up with a set of the form $\boldsymbol{a}^T\boldsymbol{x} \leq \boldsymbol{b} \cup \{y > 0\}$

Step 3: Check the satisfiability of $\boldsymbol{a}^T\boldsymbol{x} \leq \boldsymbol{b} \cup \{y > 0\}$.

Encode it as LP: maximize y subject to $\boldsymbol{a}^T\boldsymbol{x} \leq \boldsymbol{b}$.

The final system is *satisfiable* iff the *optimal value* for y is *positive*.

LP Solving Methods and Standard Form

- *Simplex* (Dantzig, 1947) — exponential time $\mathcal{O}(2^n)$, but very efficient in practice
- *Ellipsoid* (Khachiyan, 1979) — polynomial time $\mathcal{O}(n^6)$
- *Projective* (Karmarkar, 1984) — polynomial time $\mathcal{O}(n^{3.5})$

Any LP can be transformed to *standard form*:

$$
\begin{aligned}
\text{maximize} \sum_{j=1}^{n} c_j x_j \\
\text{such that} \sum_{j=1}^{m} a_{ij} x_j \leq b_i \text{ for } i = 1, ..., m \\
x_j \geq 0 \text{ for } j = 1, ..., n
\end{aligned}
$$

Example: Next, we are going to use the following running example LP:

LP Solving Methods and Standard Form [2]

$$\text{maximize } 5x_1 + 4x_2 + 3x_3$$

$$\text{such that } \begin{cases} 2x_1 + 3x_2 + x_3 \leq 5 \\ 4x_1 + x_2 + 2x_3 \leq 11 \\ 3x_1 + 4x_2 + 2x_3 \leq 8 \\ x_1, x_2, x_3 \geq 0 \end{cases}$$

Slack Variables

- Observe the first inequality: $2x_1 + 3x_2 + x_3 \leq 5$
- Define a *new variable* to represent the *slack*:

$$x_4 = 5 - 2x_1 - 3x_2 - x_3, \quad x_4 \geq 0$$

- Do this for each constraint, so that everything becomes *equalities*.
- Define a new variable to represent the *objective value*: $z = 5x_1 + 4x_2 + 3x_3$

$$\begin{aligned} &\max 5x_1 + 4x_2 + 3x_3 \\ &\text{s.t.} \begin{cases} 2x_1 + 3x_2 + x_3 \leq 5 \\ 4x_1 + x_2 + 2x_3 \leq 11 \\ 3x_1 + 4x_2 + 2x_3 \leq 8 \\ x_1, x_2, x_3 \geq 0 \end{cases} \end{aligned} \longrightarrow \begin{aligned} &\max z \\ &\text{s.t.} \begin{cases} x_4 = 5 - 2x_1 - 3x_2 - x_3 \\ x_5 = 11 - 4x_1 - x_2 - 2x_3 \\ x_6 = 8 - 3x_1 - 4x_2 - 2x_3 \\ z = 5x_1 + 4x_2 + 3x_3 \\ x_1, x_2, x_3, x_4, x_5, x_6 \geq 0 \end{cases} \end{aligned}$$

Note: Optimal solution remains optimal for the new problem.

The Simplex Strategy

- Start with a feasible solution.
  - For our example, assign 0 to all variables.
    $x_1 \mapsto 0, x_2 \mapsto 0, x_3 \mapsto 0$
  - Assign the introduced variables their computed values.
    $x_4 \mapsto 5, x_5 \mapsto 11, x_6 \mapsto 8, z \mapsto 0$
- Iteratively improve the objective value.
  - Go from $\boldsymbol{x}$ to $\boldsymbol{x}'$ only if $z(\boldsymbol{x}) \leq z(\boldsymbol{x}')$.

What can we improve here?

One option is to make x_1 larger, leaving x_2 and x_3 unchanged:

- $x_1 = 1 \quad \rightarrow \quad x_4 = 3, x_5 = 7, x_6 = 1, z = 5$ ✓
- $x_1 = 2 \quad \rightarrow \quad x_4 = 1, x_5 = 3, x_6 = 2, z = 10$ ✓
- $x_1 = 3 \quad \rightarrow \quad x_4 = -1, \ldots$ ✗ *no longer feasible!*

$$
\begin{cases}
x_4 = 5 - 2x_1 - 3x_2 - x_3 \\
x_5 = 11 - 4x_1 - x_2 - 2x_3 \\
x_6 = 8 - 3x_1 - 4x_2 - 2x_3 \\
z = 5x_1 + 4x_2 + 3x_3
\end{cases}
$$

The Simplex Strategy [2]

We can't increase x_1 *too much*. Let's increase it as much as possible, *without compromising feasibility*.

$$x_1 \mapsto 0, x_2 \mapsto 0, x_3 \mapsto 0$$

$$\begin{cases} x_4 = 5 - 2x_1 - 3x_2 - x_3 \\ x_5 = 11 - 4x_1 - x_2 - 2x_3 \\ x_6 = 8 - 3x_1 - 4x_2 - 2x_3 \\ z = 5x_1 + 4x_2 + 3x_3 \end{cases} \longrightarrow \begin{cases} x_1 \leq \frac{5}{2} \\ x_1 \leq \frac{11}{4} \\ x_1 \leq \frac{8}{3} \end{cases}$$

Select the *tightest bound*, $x_1 \leq \frac{5}{2}$.

- New assignment: $x_1 \mapsto \frac{5}{2}, x_2 \mapsto x_3 \mapsto x_4 \mapsto 0, x_5 \mapsto 1, x_6 \mapsto \frac{1}{2}, z \mapsto \frac{25}{2}$.

Now *pivot*: since x_1 became positive and x_4 became 0, swap them by isolating x_1 from the equation for x_4, then eliminating x_1 from all other equations:

The Simplex Strategy [3]

$$\begin{cases} x_4 = 5 - 2x_1 - 3x_2 - x_3 \\ x_5 = 11 - 4x_1 - x_2 - 2x_3 \\ x_6 = 8 - 3x_1 - 4x_2 - 2x_3 \\ z = 5x_1 + 4x_2 + 3x_3 \end{cases} \longrightarrow \begin{cases} x_1 = \frac{5}{2} - \frac{3}{2}x_2 - \frac{1}{2}x_3 - \frac{1}{2}x_4 \\ x_5 = 1 + 5x_2 + \qquad + 2x_4 \\ x_6 = \frac{1}{2} + \frac{1}{2}x_2 - \frac{1}{2}x_3 + \frac{3}{2}x_4 \\ z = \frac{25}{2} - \frac{7}{2}x_2 + \frac{1}{2}x_3 - \frac{5}{2}x_4 \end{cases}$$

The Simplex Strategy [4]

How can we improve z further?

- **Option 1**: decrease x_2 or x_4, but we can't since $x_2, x_4 \geq 0$.
- **Option 2**: increase x_3. *By how much?*

$$x_1 \mapsto \frac{5}{2}, x_2 \mapsto 0, x_3 \mapsto 0, x_4 \mapsto 0$$

$$\begin{cases} x_1 = \frac{5}{2} - \frac{3}{2}x_2 - \frac{1}{2}x_3 - \frac{1}{2}x_4 \\ x_5 = 1 + 5x_2 + \qquad + 2x_4 \\ x_6 = \frac{1}{2} + \frac{1}{2}x_2 - \frac{1}{2}x_3 + \frac{3}{2}x_4 \\ z = \frac{25}{2} - \frac{7}{2}x_2 + \frac{1}{2}x_3 - \frac{5}{2}x_4 \end{cases}$$

x_3's bounds: $x_3 \leq 5$, $x_3 \leq \infty$, $x_3 \leq 1$.

We increase x_3 to its tightest bound 1.

- New assignment: $x_1 \mapsto 2, x_2 \mapsto 0, x_3 \mapsto 1, x_4 \mapsto 0, x_5 \mapsto 0, x_6 \mapsto 0$.
- This gives $z = 13$, which is again an improvement.

As before, we switch x_6 and x_3, and *eliminate* x_3 from the right-hand-side:

$$\begin{cases} x_1 = \frac{5}{2} - \frac{3}{2}x_2 - \frac{1}{2}x_3 - \frac{1}{2}x_4 \\ x_5 = 1 + 5x_2 + \qquad + 2x_4 \\ x_6 = \frac{1}{2} + \frac{1}{2}x_2 - \frac{1}{2}x_3 + \frac{3}{2}x_4 \\ z = \frac{25}{2} - \frac{7}{2}x_2 + \frac{1}{2}x_3 - \frac{5}{2}x_4 \end{cases} \longrightarrow \begin{cases} x_1 = 2 - 2x_2 - 2x_4 + x_6 \\ x_5 = 1 + 5x_2 + 2x_4 \\ x_3 = 1 + x_2 + 3x_4 - 2x_6 \\ z = 13 - 3x_2 - x_4 - x_6 \end{cases}$$

The Simplex Strategy [5]

Can we improve z again?

- No, because $x_2, x_4, x_6 \geq 0$, and all *appear with negative signs* in the objective function.

So, we are done, and the optimal value of z is 13.

Optimal solution: $x_1 \mapsto 2, x_2 \mapsto 0, x_3 \mapsto 1$, with $z = 13$.

$x_1 \mapsto 2, x_2 \mapsto 0, x_3 \mapsto 1,$

$x_4 \mapsto 0, x_6 \mapsto 0$

$$\begin{cases} x_1 = 2 - 2x_2 - 2x_4 + x_6 \\ x_5 = 1 + 5x_2 + 2x_4 \\ x_3 = 1 + x_2 + 3x_4 - 2x_6 \\ z = 13 - 3x_2 - x_4 - x_6 \end{cases}$$

The Simplex Algorithm

1. Introduce *slack variables* $x_{n+1}, \ldots, x_{n+m}$ for each constraint.
2. Start with initial *feasible* solution (commonly, $x_j \mapsto 0$ for all original variables).
3. While some coefficients in the objective function are *positive*:
   - Pick a variable x_j with positive coefficient (*entering variable*).
   - Compute the tightest bound: $\min_i \left(\frac{b_i}{a_{ij}} \right)$ for $a_{ij} > 0$ (*leaving variable*).
   - *Pivot*: swap entering/leaving variables in the equation system.
4. When all coefficients are non-positive, the current solution is *optimal*. Stop.
5. Go to 4.

CDCL($\mathcal{T}$)

CDCL($\mathcal{T}$) Architecture

$$\text{CDCL}(\mathcal{T}) = \text{CDCL}(X) + \mathcal{T}\text{-solver}$$

CDCL(X):

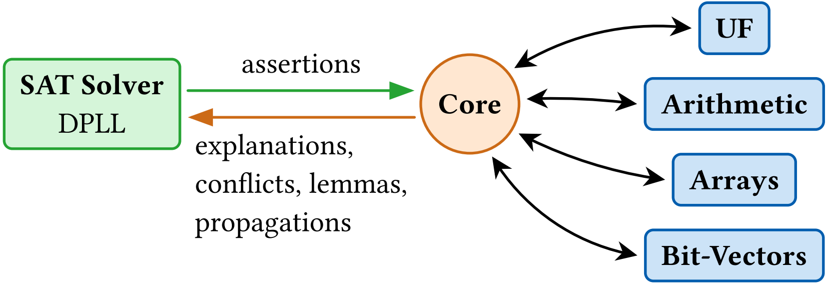
- Very *similar to a SAT solver*, enumerates Boolean models.
- Not allowed: pure literal rule (and other SAT specific heuristics).
- Required: incremental addition of clauses.
- Desirable: partial model detection.

$\mathcal{T}$-solver:

- Checks the $\mathcal{T}$-satisfiability of conjunctions of literals.
- Computes *theory propagations*.
- Produces *explanations* of $\mathcal{T}$-unsatisfiability/propagation.
- Must be *incremental* and *backtrackable*.

Typical SMT Solver Architecture

SAT Solver:
- Only sees *Boolean skeleton* of a problem.
- Builds *partial model* by assigning truth values to literals
- Sends these literals to the core as *assertions*



Theory Solvers:
- Check $\mathcal{T}$-satisfiability of sets of theory literals
- Incremental
- Backtrackable
- Conflict generation
- Theory propagation

Core:
- Sends each assertion to the appropriate theory
- Sends deduced literals to other theories/SAT solver
- Handles *theory combination*

Theory Propagation

The $\mathcal{T}$-solver does more than just check consistency — it actively *propagates* information back to the SAT solver:

- **Conflict detection:** If the current set of theory literals is $\mathcal{T}$-unsatisfiable, report a *conflict clause* (a subset of literals explaining the inconsistency).
- **Theory propagation:** If a literal ℓ is $\mathcal{T}$-entailed by the current assertions, *propagate* ℓ to the SAT solver (avoiding unnecessary case splits).
- **Lemma learning:** The $\mathcal{T}$-solver may derive useful *theory lemmas* (clauses that are $\mathcal{T}$-valid) and add them to the clause database.

Example: If the $\mathcal{T}_{\mathrm{RA}}$-solver sees $x \leq 3$ and $x \geq 5$, it immediately reports a conflict without waiting for the SAT solver to discover the contradiction.

Why incremental and backtrackable? The SAT solver frequently backtracks, so the $\mathcal{T}$-solver must efficiently *undo* assertions. Typical implementations use a *stack-based* approach: push assertions on decisions, pop on backtrack.

CDCL($\mathcal{T}$) Example

Consider the formula $\varphi = (x \doteq y \vee y \doteq z) \wedge (f(x) \not\doteq f(y) \vee y \doteq z) \wedge (x \doteq y \vee f(y) \not\doteq f(z))$ in $\mathcal{T}_{\text{EUF}}$.

Step 1: Boolean abstraction. Replace each theory atom with a Boolean variable:

- $p_1 := (x \doteq y)$, $p_2 := (y \doteq z)$, $p_3 := (f(x) \doteq f(y))$, $p_4 := (f(y) \doteq f(z))$
- Boolean skeleton: $(p_1 \vee p_2) \wedge (\neg p_3 \vee p_2) \wedge (p_1 \vee \neg p_4)$

Step 2: CDCL decides $p_1 = \text{true}$, $p_2 = \text{false}$.

- SAT solver propagates: $\neg p_3$ (from clause 2) and $\neg p_4$ (from clause 3, since $p_2 = \text{false}$).

Step 3: $\mathcal{T}$-solver checks: $\{x \doteq y, \neg(y \doteq z), \neg(f(x) \doteq f(y)), \neg(f(y) \doteq f(z))\}$.

- From $x \doteq y$, by congruence: $f(x) \doteq f(y)$. But we have $\neg(f(x) \doteq f(y))$ — **conflict!**
- Conflict clause: $\neg(x \doteq y) \vee f(x) \doteq f(y)$, *i.e.* $\neg p_1 \vee p_3$.

Step 4: CDCL learns the clause $\neg p_1 \vee p_3$, backtracks, and eventually finds SAT with $p_2 = \text{true}$.

Combining Theories

Motivation: Mixed Formulas

In practice, formulas often involve *multiple* theories simultaneously.

Example:

$$f(x) - f(y) \geq 1 \land (x \doteq y)$$

This mixes $\mathcal{T}_{\text{EUF}}$ (uninterpreted f, equality $\doteq$) and $\mathcal{T}_{\text{RA}}$ (arithmetic $-$, $\geq$, 1).

Example:

$$\text{read}(\text{write}(a, i, v), j) + 1 \leq x \land i \doteq j$$

This mixes $\mathcal{T}_{\text{AX}}$ (arrays) and $\mathcal{T}_{\text{RA}}$ (arithmetic).

A single-theory solver cannot handle such formulas. We need a *combination method* that orchestrates multiple theory solvers.

The Nelson-Oppen Method

The *Nelson-Oppen* (N-O) method combines decision procedures for *signature-disjoint*, *stably infinite* theories.

Definition 38: Two theories $\mathcal{T}_1$ and $\mathcal{T}_2$ are *signature-disjoint* if $\Sigma_1^F \cap \Sigma_2^F = \{\doteq\}$ — the only shared symbol is equality.

Definition 39: A theory $\mathcal{T}$ is *stably infinite* if every $\mathcal{T}$-satisfiable quantifier-free formula is satisfiable in a model with an *infinite* domain.

Note: Most commonly used SMT theories ($\mathcal{T}_{\mathrm{EUF}}$, $\mathcal{T}_{\mathrm{RA}}$, $\mathcal{T}_{\mathrm{IA}}$, $\mathcal{T}_{\mathrm{AX}}$) are stably infinite and pairwise signature-disjoint (when restricted to their respective sorts).

Nelson-Oppen: Purification

Step 1: Purification. Separate a mixed formula into *pure* conjuncts, one per theory.

Method: For any term t from theory $\mathcal{T}_i$ that appears as an argument in a literal of theory $\mathcal{T}_j$ (with $i \neq j$), introduce a *fresh shared variable* v, add $v \doteq t$ to $\mathcal{T}_i$'s literals, and replace t by v in $\mathcal{T}_j$'s literals.

Example: Purify $f(x) - f(y) \geq 1 \land x \doteq y$:

Introduce $v_1 \doteq f(x)$ and $v_2 \doteq f(y)$:

- $\mathcal{T}_{\text{EUF}}$: $\{v_1 \doteq f(x), v_2 \doteq f(y), x \doteq y\}$
- $\mathcal{T}_{\text{RA}}$: $\{v_1 - v_2 \geq 1\}$
- *Shared variables*: v_1, v_2 (appear in both “pure” sets)

Nelson-Oppen: Equality Propagation

Step 2: Equality propagation. The two solvers exchange *equalities and disequalities* between shared variables until a *fixed point* or a *conflict* is reached.

1. Check each pure set with its own $\mathcal{T}$-solver.
2. If either solver reports UNSAT $\Rightarrow$ the combined formula is UNSAT.
3. If the $\mathcal{T}_i$-solver deduces a new equality $v \doteq w$ between shared variables, *propagate* it to the other solver.
4. Repeat until no new equalities are deduced (*fixed point*).
5. If both solvers report SAT at the fixed point $\Rightarrow$ the combined formula is SAT.

Key insight: For *convex* theories (including $\mathcal{T}_{\text{EUF}}$ and $\mathcal{T}_{\text{RA}}$), the method only needs to propagate *equalities* — no disjunctions. This makes the procedure *deterministic* and efficient.

Definition 40: A theory $\mathcal{T}$ is *convex* if whenever $\mathcal{T} \vDash (\ell_1 \land \ldots \land \ell_n) \to (x_1 \doteq y_1 \lor \ldots \lor x_k \doteq y_k)$, then $\mathcal{T} \vDash (\ell_1 \land \ldots \land \ell_n) \to (x_i \doteq y_i)$ for some i.

Nelson-Oppen: Worked Example

Formula: $f(x) - f(y) \geq 1 \land x \doteq y$

After purification with shared variables v_1, v_2:

- $\mathcal{T}_{\text{EUF}}$: $\{v_1 \doteq f(x), v_2 \doteq f(y), x \doteq y\}$
- $\mathcal{T}_{\text{RA}}$: $\{v_1 - v_2 \geq 1\}$

Round 1:

- $\mathcal{T}_{\text{EUF}}$-solver: From $x \doteq y$ and congruence, deduces $f(x) \doteq f(y)$, hence $v_1 \doteq v_2$.
- *Propagate* $v_1 \doteq v_2$ to $\mathcal{T}_{\text{RA}}$.

Round 2:

- $\mathcal{T}_{\text{RA}}$-solver: Now has $\{v_1 - v_2 \geq 1, v_1 \doteq v_2\}$.
  - $v_1 \doteq v_2$ implies $v_1 - v_2 = 0$, contradicting $v_1 - v_2 \geq 1$.
  - **UNSAT!**

Conclusion: The original mixed formula is unsatisfiable.

SMT-LIB and Z3

SMT-LIB: The Standard Language

SMT-LIB (v2) is a standardized *input language* for SMT solvers. It defines:

- A set of *logics* (*e.g.* `QF_UF`, `QF_LIA`, `QF_LRA`, `QF_AUFLIA`, `QF_BV`, `ALL`) specifying which theories and quantifiers are allowed.
- A *command language* for interacting with solvers.
- Standard *theory declarations* shared across all solvers.

Core commands:

Command	Description
`(set-logic QF_LIA)`	Declare the logic
`(declare-sort S 0)`	Declare an uninterpreted sort with arity 0
`(declare-fun f (Int Int) Bool)`	Declare a function symbol with rank
`(define-fun g ((x Int)) Int ...)`	Define a function (macro)
`(assert (< x 5))`	Assert a formula
`(check-sat)`	Check satisfiability

SMT-LIB: The Standard Language [2]

Command	Description
`(get-model)`	Retrieve a satisfying assignment (if SAT)
`(push 1)` / `(pop 1)`	Save/restore assertion stack
`(exit)`	Terminate the session

SMT-LIB: QF_UF Example

Encoding the EUF problem: $a \doteq b \land f(a) \doteq b \land \neg(g(a) \doteq g(f(a)))$.

```
(set-logic QF_UF)
(declare-sort U 0)

(declare-fun a () U)
(declare-fun b () U)
(declare-fun f (U) U)
(declare-fun g (U) U)

(assert (= a b))
(assert (= (f a) b))
(assert (not (= (g a) (g (f a)))))

(check-sat)   ; Expected: unsat
(exit)
```

Note: In SMT-LIB, = is the built-in equality. There is no need to declare it. Variables are *declared* as zero-arity functions: `(declare-fun a () U)` means “a is a *constant* of sort U”.

SMT-LIB: QF_LIA Example

Is there an integer solution to $x + 2y \geq 5$, $x - y \leq 1$, $x \geq 0$, $y \geq 0$?

```
(set-logic QF_LIA)

(declare-fun x () Int)
(declare-fun y () Int)

(assert (>= (+ x (* 2 y)) 5))
(assert (<= (- x y) 1))
(assert (>= x 0))
(assert (>= y 0))

(check-sat)   ; Expected: sat
(get-model)   ; e.g., x = 1, y = 2
(exit)
```

SMT-LIB: QF_LIA Example [2]

Logic naming convention: `QF_` = quantifier-free. `L` = linear, `N` = non-linear. `I` = integers, `R` = reals. `A` = arrays. `UF` = uninterpreted functions. `BV` = bit-vectors. So `QF_AUFLIA` = quantifier-free arrays + UF + linear integer arithmetic.

Z3: An SMT Solver

Z3 (Microsoft Research) is one of the most widely used SMT solvers.

- Supports *all major theories*: EUF, LIA, LRA, arrays, bit-vectors, strings, datatypes, …
- Accepts *SMT-LIB* input and also has *Python*, *C/C++*, and *Java* APIs.
- Used in: Dafny, Boogie, KLEE, Rosette, angr, many other FM tools.

Example: Run the QF_LIA example with Z3 from the command line:

```
z3 example.smt2
```

Output:

```
sat
(model
  (define-fun x () Int 1)
  (define-fun y () Int 2))
```

Z3 Python API: z3py

The *z3py* library provides a Pythonic interface to Z3:

```python
from z3 import *

x, y = Ints('x y')
s = Solver()
s.add(x + 2*y >= 5)
s.add(x - y <= 1)
s.add(x >= 0, y >= 0)

if s.check() == sat:
    m = s.model()
    print(f"x = {m[x]}, y = {m[y]}")
```

Common z3py types and constructors:

- `Bool('b')`, `Int('x')`, `Real('r')` — declare sorted variables
- `Function('f', IntSort(), IntSort())` — uninterpreted function $f : \texttt{Int} \to \texttt{Int}$
- `Array('a', IntSort(), IntSort())` — array variable
- `Solver()`, `.add(...)`, `.check()`, `.model()` — solver interaction

Z3 Python API: z3py [2]

- `And(...)`, `Or(...)`, `Not(...)`, `Implies(a, b)` — logical connectives

Z3 Practical Examples

Array swap verification:

```
from z3 import *
a = Array('a', IntSort(), IntSort())
i, j = Ints('i j')
# swap a[i] and a[j]
b = Store(Store(a, i, Select(a, j)), j, Select(a, i))
# verify that b[i] == a[j] and b[j] == a[i]
s = Solver()
s.add(Not(And(Select(b, i) == Select(a, j),
              Select(b, j) == Select(a, i))))
print(s.check())  # unsat => swap is correct
```

Simple scheduling:

```
from z3 import *
A, B, C = Ints('A B C')  # start times
s = Solver()
s.add(A >= 0, B >= 0, C >= 0)   # non-negative start
s.add(A + 3 <= B)               # A finishes before B starts
```

Z3 Practical Examples [2]

```
s.add(B + 2 <= C)                  # B finishes before C starts
s.add(C + 1 <= 8)                  # C finishes by deadline 8
if s.check() == sat:
    print(s.model())  # e.g., A=0, B=3, C=5
```

Advanced Applications of SMT

Symbolic Execution

Symbolic execution explores program paths using *symbolic* rather than concrete inputs, building *path conditions* as SMT formulas.

Example:

```
def abs(x: int) -> int:
    if x >= 0: return x
    else: return -x
```

- Path 1: condition $x \geq 0$, result $= x$.
- Path 2: condition $x < 0$, result $= -x$.
- To find an input where `abs(x) < 0`: query SMT for $(x \geq 0 \land x < 0) \lor (x < 0 \land -x < 0)$ — UNSAT!

Tools: KLEE (LLVM), SAGE (Microsoft), angr (binary analysis). All rely on SMT solvers (typically Z3) as the constraint-solving backend.

CEGAR and CEGIS

Counterexample-Guided Abstraction Refinement (CEGAR):

1. Start with a *coarse* abstraction of the program.
2. Check if the abstraction satisfies the property (using an SMT solver).
3. If a counterexample is found, check if it is *spurious* (infeasible in the concrete program).
4. If spurious, *refine* the abstraction and repeat.

Counterexample-Guided Inductive Synthesis (CEGIS):

1. Propose a candidate program (from a template).
2. Check if it satisfies the specification for *all* inputs (using an SMT solver).
3. If a counterexample input is found, add it to the test suite and re-synthesize.

Common pattern: both CEGAR and CEGIS use SMT solvers in a *verify–refine loop*. The solver alternates between finding counterexamples and checking candidates.

Syntax-Guided Synthesis (SyGuS)

Definition 41 (SyGuS Problem): Given a *specification* $\varphi(x, f(x))$ and a *grammar* G defining the space of candidate expressions, find a function f expressible in G such that $\forall x.\, \varphi(x, f(x))$.

Example: **Synthesize** $\max(x, y)$:

- Spec: $f(x, y) \geq x \land f(x, y) \geq y \land (f(x, y) = x \lor f(x, y) = y)$
- Grammar: $f \mathrel{::=} x \mid y \mid \texttt{ite}(f \geq f, f, f)$
- Solution: $\texttt{ite}(x \geq y, x, y)$

SyGuS competitions (since 2014) drive solver development. Key solvers: **CVC5** (built-in SyGuS), **EUSolver**, **DryadSynth**.

The big picture: SMT solvers are not just decision procedures — they are the *engine* behind program analysis, synthesis, and verification. Everything from Dafny to KLEE to CVC5's synthesizer relies on efficient SMT solving.

Exercises

Exercise: Theory Satisfiability

Determine whether the following sets of literals are $\mathcal{T}$-satisfiable or $\mathcal{T}$-unsatisfiable. If satisfiable, provide a $\mathcal{T}$-interpretation. If unsatisfiable, show why.

1. In $\mathcal{T}_{\mathrm{EUF}}$: $\{a \doteq b,\ b \doteq c,\ f(a) \not\doteq f(c)\}$
2. In $\mathcal{T}_{\mathrm{RA}}$: $\{x + y \leq 3,\ 2x - y \geq 5,\ x \leq 1\}$
3. In DL: $\{x - y \leq 2,\ y - z \leq 3,\ z - x \leq -6\}$

Exercise: Nelson-Oppen and SMT-LIB

1. Purify the following formula into $\mathcal{T}_{\text{EUF}}$ and $\mathcal{T}_{\text{RA}}$ components:

$$f(x+1) \doteq f(y) \land x - y \geq 0 \land x \leq y$$

   Run the Nelson-Oppen equality propagation. Is the formula satisfiable?

2. Encode the following problem in SMT-LIB (`QF_LIA`): "Find integers x, y, z such that $x + y + z = 15$, $x \geq 1$, $y \geq 1$, $z \geq 1$, and $x \leq y \leq z$."

3. Write a z3py script to verify that for all integers x: if $x > 0$, then $x + x > x$. *(Hint: show the negation is UNSAT.)*