Automated Reasoning

Luca Panariello

November 30, 2024

Automated Reasoning is the study of algorithms and systems that allow computers to reason about logical statements.

In this chapter, we will introduce the syntax and semantics of First-Order Logic (FOL), which is the most widely used logic in the field of Automated Reasoning.

We will also introduce the concept of a *model* and the notion of *validity* of a logical statement.

Automated Reasoning is achieved by using symbol reasoning, which is the manipulation of symbols according to the rules of logic.

Contents

Chapter 1

Propositional Logic

Propositional logic is a formal system that deals with propositions, which are statements that are either true or false.

1.1 Syntax of Propositional Logic

It is composed of a set $\mathcal{V}$ of symbols called *propositional variables* which are denoted by $P, Q, R, \ldots$, $x, y, z, \ldots$, $1, 2, 3, \ldots$ or $true, false$.

An atomic variable are:

- $\top$ which is always true.
- $\bot$ which is always false.
- P which is a propositional variable.

The logical connectives are $\neg$ (negation), $\wedge$ (conjunction), $\vee$ (disjunction), $\implies$ (implication) and $\iff$ (equivalence).

Definition 1.1.1: Propositional Formula

A **propositional formula** is defined as follows:

- Every atomic variable is a formula.
- If F and G are formulas, then $\neg F$, $F \wedge G$, $F \vee G$, $F \implies G$, $F \iff G$ are formulas.

Definition 1.1.2: Propositional Interpretation

A **propositional interpretation** of a formula is a function that assigns a truth value to each atomic variable, and it is defined as follows: $\mathcal{I} : \mathcal{V} \to \{true, false\}$.

1.2 Semantic in Propositional Logic

An interpretation $\mathcal{I}$ is said to *satisfy* a formula F, written as $\mathcal{I} \vDash F$:

- if F is an atomic variable P, then $\mathcal{I}(P) = true$.
- if $F = \neg G$, then $\mathcal{I} \vDash F$ if $\mathcal{I} \nvDash G$.
- if $F = G \wedge H$, then $\mathcal{I} \vDash F$ if $\mathcal{I} \vDash G$ and $\mathcal{I} \vDash H$.
- if $F = G \vee H$, then $\mathcal{I} \vDash F$ if $\mathcal{I} \vDash G$ or $\mathcal{I} \vDash H$.
- if $F = G \implies H$, then $\mathcal{I} \vDash F$ if $\mathcal{I} \nvDash G$ or $\mathcal{I} \vDash H$.
- if $F = G \iff H$, then $\mathcal{I} \vDash F$ if $\mathcal{I} \vDash G$ and $\mathcal{I} \vDash H$ or $\mathcal{I} \nvDash G$ and $\mathcal{I} \nvDash H$.

Definition 1.2.1: Satisfiable Formula

A formula F is **satisfiable** if $\exists \mathcal{I}$ interpretation such that $\mathcal{I} \vDash F$.

Definition 1.2.2: Valid Formula

A formula F is **valid** if $\forall \mathcal{I}$ interpretation such that $\mathcal{I} \vDash F$.

Remark 1.2.3: Unsatisfiable and Invalid Formulas

A formula F is **unsatisfiable** if $\forall \mathcal{I}$ interpretation such that $\mathcal{I} \nvDash F$. A formula F is **invalid** if $\exists \mathcal{I}$ interpretation such that $\mathcal{I} \nvDash F$.

Remark 1.2.4: Implication of Validity

Let F be a formula, than we can observe that:

F		$\neg F$
Satisfiable	$\implies$	Invalid
Valid	$\implies$	Unsatisfiable
Invalid	$\implies$	Satisfiable
Unsatisfiable	$\implies$	Valid

[IDEA OF THE PROBLEM OF SATISFIABILITY IN PROPOSITIONAL LOGIC] [NOTE IN INTRODUZIONE 2 OTTOBRE]

Given a formula F, than F is finite, hence the number of propositional variables is finite, therefore the number of interpretations is finite.

In particular the number of interpretations is 2^n, where n is the number of propositional variables.

[SCHEMA AD ALBERO] [INTRODUZIONE 2 OTTOBRE]

testing every interpretation is costly and inefficient, we need to design a decision procedure that is able to determine the satisfiability of a formula in a more efficient way, using normal forms.

[DECISION PROCEDURE] [INTRODUZIONE 2 OTTOBRE]

1.3 Normal Forms

1.3.1 Negation Normal Form

Definition 1.3.1: Negation Normal Form

A formula F is in **negation normal form** if the only connective that appears is $\neg, \wedge, \vee$ and $\neg$ is only applied to atomic variables.

The procedure to convert a formula F into negation normal form is as follows:

- $\neg\neg G \equiv G$.
- $\neg(G \wedge H) \equiv \neg G \vee \neg H$.
- $\neg(G \vee H) \equiv \neg G \wedge \neg H$.
- $(G \implies H) \equiv \neg G \vee H$.

- $(G \iff H) \equiv (\neg G \lor H) \land (G \lor \neg H)$

1.3.2 Disjunctive Normal Form

Definition 1.3.2: Disjunctive Normal Form

A formula F is in **disjunctive normal form** if it is a disjunction of conjunctions of atomic variables.

$$F = D_1 \lor D_2 \lor \cdots \lor D_n \tag{1.1}$$

where $D_i = (L_1^i \land L_2^i \land \cdots \land L_n^i)$ is a conjunction of atomic variables called *cube*.

The procedure to convert a formula F into disjunctive normal form is to convert it into negation normal form and then apply the distributive law:

- $G \land (H \lor K) = (G \land H) \lor (G \land K)$.
- $(G \lor H) \land K = (G \land K) \lor (H \land K)$.

1.3.3 Conjunctive Normal Form

Definition 1.3.3: Conjunctive Normal Form

A formula F is in **conjunctive normal form** if it is a conjunction of disjunctions of atomic variables.

$$F = C_1 \land C_2 \land \cdots \land C_n \tag{1.2}$$

where $C_i = (L_1^i \lor \cdots \lor L_n^i)$ is a disjunction of atomic variables called *clause*.

The procedure to convert a formula F into conjunctive normal form is to convert it into negation normal form and then apply the distributive law:

- $G \lor (H \land K) = (G \lor H) \land (G \lor K)$.
- $(G \land H) \lor K = (G \lor K) \land (H \lor K)$.

For the SAT problem, the conjunctive normal form is the most used normal form, because it is the most efficient to determine the satisfiability of a formula.

The distributive law cause the formula to grow exponentially, hence adding cost to the decision procedure.

Chapter 2

First-Order Logic

First Order Logic is a formal system that deals with logical statements that are more complex than propositional logic.

It is composed of a set of symbols that are used to represent logical statements, and a set of rules that define how these symbols can be combined to form logical statements.

It introduce the concept of *variables*, *functions* and *predicates*, which allows us to reason about objects and relations between objects.

2.1 Syntax in First-Order Logic

Definition 2.1.1: Signature

A **signature** is a tuple $\Sigma = (\mathcal{C}, \mathcal{F}, \mathcal{P})$ where:

- $\mathcal{C}$ is a set of constant symbols.
- $\mathcal{F}$ is a set of function symbols.
- $\mathcal{P}$ is a set of predicate symbols.

The **constant** symbols denote individual elements (e.g. 0, 1, a, b, $\dots$).

The **function** symbols denote functions that take a number of arguments and return a value (e.g. $+$, $\times$, f, g, $\dots$).

The **predicate** symbols denote relations that take a number of arguments and return a truth value (e.g. $=$, $<$, P, Q, $\dots$).

The elements of $\mathcal{C}$, $\mathcal{F}$ and $\mathcal{P}$ are called *symbols*.

The arity of a function or predicate symbol is the number of arguments it takes.

The difference between a function and a predicate lies in their connection, infact a function returns a value while a predicate returns a truth value.

Example 2.1.2: Predicate vs Function

$f(x) = f(y)$ these functions are connected by the equality predicate $=$, while these predicates $P(x) \iff P(y)$ are connected by logical equivalence $\iff$

Remark 2.1.3: Functions of Predicate

A predicate P can be seen as a function f_P that returns a truth value. So introducing a constant symbol $\circ$ witch denotes "truth" than we can define f_P as:

$$f_P(x_1, \dots, x_n) = \circ \quad \text{if } P(x_1, \dots, x_n) \text{ is true}$$

where $\circ$ is added to the signature. $\Sigma' = \Sigma \cup \{\circ\}$

In FOL, there are logical connectives that are used to combine logical statements: $\neg$ (negation), $\wedge$ (conjunction), $\vee$ (disjunction), $\implies$ (implication), $\iff$ (equivalence).

Using a signature Σ and a set of variables $\mathcal{X}$, we can define the syntax of FOL.

Definition 2.1.4: Term

Let Σ be a signature and $\mathcal{X}$ a set of variables. A **term** is defined as follows:

- Every constant symbol $c \in \mathcal{C}$ is a term.
- Every variable $x \in \mathcal{X}$ is a term.
- Every n-ary function symbol $f \in \mathcal{F}$ with $t_1, \dots, t_n$ are terms, then $f(t_1, \dots, t_n)$ is a term.

Definition 2.1.5: Atom

Let Σ be a signature and $\mathcal{X}$ a set of variables. An **atom** is defined as follows:

- Every n-ary predicate symbol $P \in \mathcal{P}$ with $t_1, \dots, t_n$ are terms, then $P(t_1, \dots, t_n)$ is an atom.

Definition 2.1.6: Literal

A **literal** is an atom or the negation of an atom.

Definition 2.1.7: Formula

Let Σ be a signature and $\mathcal{X}$ a set of variables. A **formula** is defined as follows:

- Every atom is a formula.
- If F and G are formulas, then $\neg F$, $F \wedge G$, $F \vee G$, $F \implies G$, $F \iff G$ are formulas.
- If F is a formula and $x \in \mathcal{X}$ is a variable, then $\forall x.F$ and $\exists x.F$ are formulas.

From a formula we can destinguish the *free variables* and the *bound variables*. The free variables are the variables that are not bounded by a quantifier, while the bound variables are.

Example 2.1.8: Free and Bound Variables

Give the formula H:

$$f(x) = b \wedge \forall y.f(y) = b \implies f(f(y)) = b$$

We can define the free variables as $FV(H) = \{x\}$ and the bound variables as $BV(H) = \{y\}$. While b is a constant symbol.

Remark 2.1.9: Renaming Variables

Remark that a variable cannot be both free and bound at the same time. There is a issue with the renaming of variables: free variables cannot be renamed, while bound variables can, but the it cannot be renamed to a variable that is already in the formula.

[IMMAGINE VALIDITY PROBLEM IN FOL][NOTE IN SIGNATURE 3 OTTOBRE]

2.2 Semantics in First-Order Logic

Definition 2.2.1: Interpretation

Let Σ be a signature. An **interpretation** $\mathcal{I}$ of Σ is a tuple $\mathcal{I} = (\mathcal{D}, \Phi)$ where:

- $\mathcal{D}$ is a non-empty set called the *domain* of $\mathcal{I}$.
- Φ is a function that assigns to each symbol in Σ an element of $\mathcal{D}$.

The function Φ is defined as follows:

- $\forall c \in \mathcal{C}$, $\Phi(c) \in \mathcal{D}$.
- $\forall f \in \mathcal{F}$, $\Phi(f) : \mathcal{D}^n \to \mathcal{D}$.
- $\forall P \in \mathcal{P}$, $\Phi(P) : \mathcal{D}^n$.

Where n is the arity of the function or predicate symbol.

So an interpretation assigns a meaning to the symbols in the signature, there can be multiple interpretations for the same signature.

Example 2.2.2: Interpretation of Integers

Let $\Sigma = (\{a, b\}, \{f\}, \{R\})$ be a signature. An interpretation $\mathcal{I}$ of Σ can be defined as follows:

- $\mathcal{D} = \mathbb{Z}$.
- $\Phi(a) = -3$.
- $\Phi(b) = 3$.
- $\Phi(f) = + : \mathbb{Z}^2 \to \mathbb{Z}$ is the addition function.
- $\Phi(R) = \geq : \mathbb{Z}^2$ is the greater than or equal to predicate.

Example 2.2.3: Interpretation of Color

Let $\Sigma = (\{a, b, c\}, \emptyset, \{R\})$ be a signature. An interpretation $\mathcal{I}$ of Σ can be defined as follows:

- $\mathcal{D} = \{red, green, blue\}$.
- $\Phi(a) = red$.
- $\Phi(b) = blue$.
- $\Phi(c) = green$.
- $\Phi(R) = \{(red, blue), (red, red)\}$.

A term t is evaluated in an interpretation $\mathcal{I}$ as follows:

$$[t]_{\mathcal{I}} = \begin{cases} \Phi(c) & \text{if } t = c \in \mathcal{C} \text{ is a constant symbol} \\ \Phi(f)([t_1]_{\mathcal{I}}, \ldots, [t_n]_{\mathcal{I}}) & \text{if } t = f(t_1, \ldots, t_n) \in \mathcal{F} \text{ is a function symbol} \\ \beta(x) & \text{if } t = x \in \mathcal{X} \text{ is a variable} \end{cases}$$

Where n is the arity of the function symbol f and β is an assignment function that assigns a value to a variable.

2.3 Satifaction and Validity in First-Order Logic

A formula F is evaluated in an interpretation $\mathcal{I}$ and it is said to be *satisfied* in $\mathcal{I}$ if it evaluates to true, noted as $\mathcal{I} \models F$, otherwise it is said to be *unsatisfied* in $\mathcal{I}$, noted as $\mathcal{I} \not\models F$.

The first-order formulas follows the same definition of satisfaction as the propositional formulas:

- **Satisfiable Formula** (1.2.1).
- **Valid Formula** (1.2.2).
- **Unsatisfiable Formula** (1.2.3).
- **Invalid Formulas** (1.2.3).
- **Implication of Validity** (1.2.4).

But the satisfiable relation for first-order formulas are defined as follows: Let F, G, H be formulas and $\mathcal{I}$ be an interpretation, than:

- $\mathcal{I} \models \neg F$ if $\mathcal{I} \not\models F$.
- $\mathcal{I} \models F \wedge G$ if $\mathcal{I} \models F \wedge \mathcal{I} \models G$.
- $\mathcal{I} \models F \vee G$ if $\mathcal{I} \models F \vee \mathcal{I} \models G$.
- $\mathcal{I} \models F \implies G$ if $\mathcal{I} \not\models F \implies \mathcal{I} \models G$.
- $\mathcal{I} \models F \iff G$ if $\mathcal{I} \models F \iff \mathcal{I} \models G$.
- $\mathcal{I} \models \forall x.F$ if $\forall d \in \mathcal{D} : \mathcal{I} \models_{\beta[x \to d]} G$
- $\mathcal{I} \models \exists x.F$ if $\exists d \in \mathcal{D} : \mathcal{I} \models_{\beta[x \to d]} G$

Where $\mathcal{I} \models_{\beta[x \to d]} G$ means that the formula G is satisfied in $\mathcal{I}$ with the assignment function β that assigns the value d to the variable x.

$$\beta[x \to d](y) = \begin{cases} d & \text{if } y = x \\ \beta(y) & \text{otherwise} \end{cases} \quad \forall y \in \mathcal{X}$$

2.4 Logical Consequence

Logical Consequence is the relation between a set of formulas (assumptions) and a formula (conjecture), where the conjecture is true if the assumptions are true.

Let H be a set of formulas, called "*assumption*", and φ be a formula, called "*conjecture*":

We have that $H \models \varphi$ or equivalently $\models H \implies \varphi$, which means that φ is **logica consequence** of H, then:

$$\forall \mathcal{I} \text{ interpretation } : \mathcal{I} \models H \iff \mathcal{I} \models \varphi \iff H \cup \{\neg\varphi\} \text{ is unsatisfiable}$$

The last coimplication derives from the fact that $\neg\varphi$ is the negation of the conjecture, so if we find an interpretation that satisfies all the formulas in H then it must satisfy the conjecture, making the satisfaction of $\neg\varphi$ impossible.

We need a way to determine if a formula is a logical consequence of a set of formulas: we can build a decision procedure that checks if the set of formulas is unsatisfiable, in particular the procedure checks if $H \cup \{\neg\varphi\} \models \bot$.

[IMMGINE CON PROCEDURA iN 8 OTTOBRE]

Chapter 3

First-Order Theories

3.1 Theories in First-Order Logic

A **theory** formalize structures in a specific domain of interest, and help us reason about the properties of these structures. It really useful in verification.

Will be introduced some definitions that concerns theories in FOL.

Definition 3.1.1: Theory

A **theory** $\mathcal{T}$ is definted as a tuple $\mathcal{T} = (\Sigma, \mathcal{A})$ where:

- Σ is a signature.
- $\mathcal{A}$ is a set of formulas called *axioms*, with only elements of the signature.

Definition 3.1.2: Sigma-Forumla

A formula F is a **Σ-formula** if it contains symbols in the signature Σ, as well as the logical connectives, quantifiers and variables.

Definition 3.1.3: Theory-Interpretation

If $\mathcal{I}$ is an interpretation of Σ, then $\mathcal{I}$ is a **$\mathcal{T}$-interpretation** of a theory $\mathcal{T} = (\Sigma, \mathcal{A})$ if $\mathcal{I} \models \mathcal{A}$.

Definition 3.1.4: Theory-Satisfiable Formula

Let $\mathcal{T} = (\Sigma, \mathcal{A})$ be a theory. And F be a Σ-formula. If $\exists \mathcal{I}$ interpretation of Σ:

$$\mathcal{I} \vDash \mathcal{A} \wedge \mathcal{I} \vDash F$$

Which means that F is satisfied in $\mathcal{I}$ and $\mathcal{I}$ is a $\mathcal{T}$-interpretation.
So F is **$\mathcal{T}$-satisfiable** in the theory $\mathcal{T}$.

Definition 3.1.5: Theory-Vaild Formula

Let $\mathcal{T} = (\Sigma, \mathcal{A})$ be a theory. And F be a Σ-formula. If $\forall \mathcal{I}$ interpretation of Σ:

$$\mathcal{I} \vDash \mathcal{A} \implies \mathcal{I} \vDash F$$

Which means F is valid ($\vDash F$) in the theory $\mathcal{T}$ if every $\mathcal{T}$-interpretation satisfies F.
Then F is a **$\mathcal{T}$-valid formula**, also noted as $\mathcal{T} \vDash F$.

Definition 3.1.6: Theory Fragment

A **theory fragment** is a theory that deals only with a subset of formulas of the original theory.

Definition 3.1.7: Quantifier-Free Fragment

The **quantifier-free fragment** of a theory $\mathcal{T}$ is the theory that contains only the formulas that do not containg quantifier: $\forall$ and $\exists$. Which means that the variables in the formulas are free.

3.2 Theory of Equality

The theory of equality is a theory that is centered around the **equality predicate** $\simeq$ and the equivalence axioms.

Definition 3.2.1: Equivalence Axioms

Let $\mathcal{X}$ be a set of variables and $\simeq$ be a predicate symbol. Let $\mathcal{A}$ be a set of formulas called **equivalence axioms** if it contains the following formulas:

- **Reflexivity**: $\forall x \in \mathcal{X}. \quad x \simeq x.$
- **Symmetry**: $\forall x, y \in \mathcal{X}. \quad x \simeq y \implies y \simeq x.$
- **Transitivity**: $\forall x, y, z \in \mathcal{X}. \quad x \simeq y \wedge y \simeq z \implies x \simeq z.$

Definition 3.2.2: Congruence Axioms

Let $\mathcal{X}$ be a set of variables and $\simeq$ be a predicate symbol. Let $\mathcal{F}$ be a set of function symbols and $\mathcal{P}$ be a set of predicate symbols. Let $\mathcal{A}$ be a set of formulas called **congruence axioms** if it contains the following formulas:

- **Function Congruence**: $\forall$ function symbol $f \in \mathcal{F}$ with arity n and $\forall x_1, \ldots, x_n, y_1, \ldots, y_n \in \mathcal{X}$:

$$x_1 \simeq y_1 \wedge \cdots \wedge x_n \simeq y_n \implies f(x_1, \ldots, x_n) \simeq f(y_1, \ldots, y_n)$$

- **Predicate Congruence**: $\forall$ predicate symbol $R \in \mathcal{P}$ with arity n and $\forall x_1, \ldots, x_n, y_1, \ldots, y_n \in \mathcal{X}$:

$$x_1 \simeq y_1 \wedge \cdots \wedge x_n \simeq y_n \implies R(x_1, \ldots, x_n) \iff R(y_1, \ldots, y_n)$$

Which states that if the arguments of a function or predicate are equal, then the result of the function or the truth value of the predicate is equal.

Definition 3.2.3: Theory of Equality

Let $\Sigma_E = (\mathcal{C}, \mathcal{F}, \mathcal{P} \cup \{\simeq\})$ be a signature. Let $\mathcal{A}_E$ be a set of formulas called **equality axioms** if it contains the Equivalence Axioms and the Congruence Axioms. The it can be defined the **theory of equality** as:

$$\mathcal{T}_E = (\Sigma_E, \mathcal{A}_E)$$

Notation 3.2.4: Disequaliti in Theory of Equality

It is possible to define the **disequality predicate** $\not\simeq$ as:

$$x \not\simeq y \iff \neg(x \simeq y)$$

Where x and y are variables in $\mathcal{X}$. It also possibleto ridefine the equality signature as:

$$\Sigma_E = (\mathcal{C}, \mathcal{F}, \mathcal{P} \cup \{\simeq, \not\simeq\})$$

Remark 3.2.5: Theory of Equality

Since the equality axioms contanin the equivalence axioms and the congruence axioms, then the equality predicate $\simeq$ is a **congruence relation**.

Example 3.2.6: Satisfiability in Theory of Equality

The formula $a \simeq b \wedge f(a) \simeq f(b)$ is satisfiable in the theory of equality ($\mathcal{T}_E$-satisfiable). While the formula $a \simeq b \wedge R(a) \iff \neg R(b)$ is unsatisfiable in the theory of equality ($\mathcal{T}_E$-unsatisfiable).

Notation 3.2.7: Nested Function

The notation $f^{(n)}(x)$ denotes the n-th iteration of the function f on the argument x:

$$f^{(n)}(x) = \underbrace{f(f(\dots f(x)\dots))}_{n}$$

Example 3.2.8: Human Reasoning for Satisfiability

Let the formula:

$$F = \underbrace{f^{(3)}(x) \simeq x}_{\text{EQ1}} \land \underbrace{f^{(5)}(x) \simeq x}_{\text{EQ2}} \land \underbrace{f(x) \not\simeq x}_{\text{EQ3}}$$

Using human reasoning we can see that:

- EQ1 + EQ2 $\implies f^{(2)}(x) \simeq x$ (EQ4).
- EQ1 + EQ4 $\implies f(x) \simeq x$ (EQ5).
- EQ3 + EQ5 $\implies \bot$.

So the formula F is unsatisfiable.

The human reasoning is not efficient for large formulas, and for machines, we need a decision procedure that can determine the satisfiability with an algorithm that can be executed by a computer.

3.3 Theory of Lists

The theory of lists is a theory that is centered around the **list data structure** and the operations that can be performed on lists.

This theory is based on the theory of equality.

Definition 3.3.1: List Signature

The **list signature** Σ_L is a signature 2.1.1 defined as:

$$\Sigma_L = (\emptyset, \{\text{cons}, \text{car}, \text{cdr}\}, \{\text{atom}, \simeq\})$$

- cons is the constructor function.
- car (Content of Address Register) is the function that returns the first element of the list.
- cdr (Content of Decrement Register) is the function that returns the rest of the list.
- atom is the predicate that checks if the list is empty.
- $\simeq$ is the equality predicate.

Definition 3.3.2: List Axioms

Let $\mathcal{X}$ be a set of variables and Σ_L be the list signature. The **list axioms** $\mathcal{A}_L$ are defined as:

- **Equivalence Axioms** (3.2.1) on the equality predicate $\simeq$ and variables in $\mathcal{X}$.
- **Function Congruence** (3.2.2) on functions cons, car and cdr.
- **Predicate Congruence** (3.2.2) on the predicate atom.
- **Left and Right Projection**: $\forall x, y \in \mathcal{X}$.

$$\text{car}(\text{cons}(x, y)) \simeq x \wedge \text{cdr}(\text{cons}(x, y)) \simeq y$$

- **Construction Axiom**: $\forall x \in \mathcal{X}$.

$$\neg\text{atom}(x) \implies x \simeq \text{cons}(\text{car}(x), \text{cdr}(x))$$

- **Atom Axiom**: $\forall x, y \in \mathcal{X} : \quad \neg\text{atom}(\text{cons}(x, y))$

Remark 3.3.3: Extensionalty on Lists

The *Function Congruence* axioms and the *Predicate Congruence* axioms imply that two lists are equal if and only if they have the same elements in the same order:

$$x \simeq y \iff \text{car}(x) \simeq \text{car}(y) \wedge \text{cdr}(x) \simeq \text{cdr}(y) \quad \forall x, y \in \mathcal{X}$$

This is called the **extensionality** property of lists.

Definition 3.3.4: Theory of Lists

The **theory of lists** $\mathcal{T}_L$ is defined as:

$$\mathcal{T}_L = (\Sigma_L, \mathcal{A}_L)$$

Remark 3.3.5: Decidability of the Theory of Lists

The theory of lists is not decidable, in general, but the quantifier-free fragment of the theory of lists is decidable.

3.3.1 Theory of Acyclic Lists

The theory of acyclic lists is a theory that is centered around the **acyclic list data structure** this type of list does not contain any recursion in the list structure.

New axioms are added to the theory of lists to ensure that the lists are acyclic.

Definition 3.3.6: Acyclic List Axioms

Let $\mathcal{X}$ be a set of variables and Σ_L be the list signature. The **acyclic list axioms** $\mathcal{A}_L^+$ are defined as: For all variables $x \in \mathcal{X}$ and for all $t[x]$ terms that contain x:

$$x \simeq t[x] \implies \text{car}(x) \not\simeq x \wedge \text{cdr}(x) \not\simeq \text{cdr}(x) \wedge \text{car}(\text{cdr}(x)) \not\simeq x$$

And contains the $\mathcal{A}_L$ axioms.

In this way, the acyclic list cannot contain their own address in the list.

Definition 3.3.7: Theory of Acyclic Lists

The **theory of acyclic lists** $\mathcal{T}_L^+$ is defined as:

$$\mathcal{T}_L^+ = (\Sigma_L, \mathcal{A}_L^+)$$

3.3.2 Theory of Lists with Specified Atoms

The theory of lists with specified atoms is a theory that is centered around the **list data structure** and the operations that can be done on atomic lists.

In this case the an additional axiom is added to the theory of lists:

Definition 3.3.8: Specified Atom Axiom

Let $\mathcal{X}$ be a set of variables and Σ_L be the list signature. The **specified atom axiom** $\mathcal{A}_L^{\text{atom}}$ is defined as:

$$\text{atom}(x) \implies \text{atom}(\text{car}(x)) \wedge \text{atom}(\text{cdr}(x))$$

And contains the $\mathcal{A}_L$ axioms.

Definition 3.3.9: Theory of Lists with Specified Atoms

The **theory of lists with specified atoms** $\mathcal{T}_L^{\text{atom}}$ is defined as:

$$\mathcal{T}_L^{\text{atom}} = (\Sigma_L, \mathcal{A}_L^{\text{atom}})$$

3.3.3 Theory of Possibly Empty Lists

The theory of possibly empty lists is a theory that is centered around the **list data structure** and the operations that can be done on possibly empty lists.

In this case, an additional axiom is added to the theory of lists:

Definition 3.3.10: Possibly Empty List Signature

The **possibly empty list signature** Σ_L^{nil} is defined as:

$$\Sigma_L^{\text{nil}} = (\{\text{nil}\}, \{\text{cons}, \text{car}, \text{cdr}\}, \{\text{atom}, \simeq\})$$

Where nil is the constant that represents the empty list. And the other symbols are the same as the list signature 3.3.1.

Definition 3.3.11: Possibly Empty List Axiom

Let $\mathcal{X}$ be a set of variables and Σ_L^{nil} be the list signature. The **possibly empty list axiom** $\mathcal{A}_L^{\text{nil}}$ is defined as:

- $\text{car}(\text{nil}) \simeq \text{nil} \wedge \text{cdr}(\text{nil}) \simeq \text{nil}$
- $x \simeq \text{nil} \implies x \simeq \text{cons}(\text{car}(x), \text{cdr}(x)) \quad \forall x \in \mathcal{X}$

And contains the $\mathcal{A}_L$ axioms.

Definition 3.3.12: Theory of Possibly Empty Lists

The **theory of possibly empty lists** $\mathcal{T}_L^{\text{nil}}$ is defined as:

$$\mathcal{T}_L^{\text{nil}} = (\Sigma_L^{\text{nil}}, \mathcal{A}_L^{\text{nil}})$$

Chapter 4

QF-Fragment of the Theory of Equality

Non all the Logic formulas can be decided by a decision procedure:

- **PL** (Propositional Logic) is decidable problem.
- **Fragment of FOL** is decidable problem.
- **FOL** (First-Order Logic) formula is valid is a semi-decidable problem.
- **FOL** formula is invalid is a non-semi-decidable problem.
- **HOL** (Higher-Order Logic) is undecidable problem.

In this paper we will focus on the **PL** and **FOL** formulas.

In particular we will focus on the **QF-Fragment of the Theory of Equality** (QF-$\mathcal{T}_E$).

4.1 Formulas in QF-Fragment of the TOE

The QF-$\mathcal{T}_E$ is the fragment of the first-order logic that contains only quantifier-free formulas and the equality predicate.

The idea is to restrict the formulas to a certain form in order to make the problem decidable by the same decision procedure.

This "*prepocessing*" can be done with the following steps:

1. Remove all the quantifiers.

2. Tranform the formula in NNF (1.3.1).

3. Tranform the formula in DNF (1.3.2).

4. Removing all other predicate symbols except the equality predicate (2.1.3)

In this way the formula is in the QF-$\mathcal{T}_E$, and it can be expressed as a conjunction of equalities and disequalities.

In this way if a block of the formula is false, the whole formula is unsatisfiable.

The QF-$\mathcal{T}_E$-Formula F can also be expressed as a set of equations and disequations:

$$F = \{s_i \simeq t_i\}_{i=1}^{n} \cup \{s_j \not\simeq t_j\}_{j=n+1}^{n+m}$$

Where s_i, t_i, s_j, t_j are terms.

4.2 Congruence Closure of a Binary Relation

Definition 4.2.1: Binary Relation

A **binary relation** R is a subset of the cartesian product of a set S and itself.

$$R \subseteq S \times S$$

Definition 4.2.2: Equivalence Relation

A binary relation R is an **equivalence relation** if it satisfies the *equivalence axioms* 3.2.1 for the predicate.

Definition 4.2.3: Congruence Relation

A equivalence relation R is a **congruence relation** if it satisfies the the *congruence axioms* 3.2.2 for the predicate.

Definition 4.2.4: Equivalence Class

Given a set S and an equivalence relation R on S, the **equivalence class** of an element $s \in S$ is the set:

$$[s]_R = \{t \in S \mid sRt\}$$

Example 4.2.5: Modulo 2 Equivalence Class

Given the set $S = \mathbb{Z}$ and the equivalence relation R defined as:

$$sRt \iff s \equiv t \mod 2 \iff s \equiv_2 t$$

The equivalence class of 0 is:

$$[0]_{\equiv_2} = \{\dots, -4, -2, 0, 2, 4, \dots\}$$

Definition 4.2.6: Refinement

Let R and R' be two equivalence relations on a set S. The relation R is a **refinement** of R' if:

$$\forall s, t \in S \; sRt \implies sR't$$

Which means $R \subsetneq R'$, and it is denoted as $R \sqsubseteq R'$.

Remark 4.2.7: Refinement

We can say that the relation R is a refinement of the relation R'. But R' is not a refinement of R, in fact:

$$\forall s, t \in S \; sR't \not\Rightarrow sRt$$

Example 4.2.8: Refinement of Modulo 2 Class

We can say that the relation $\equiv_4$ is a refinement of the relation $\equiv_2$.

$$\forall s, t \in \mathbb{Z} \; s \equiv_4 t \implies s \equiv_2 t$$

[IMAGE OG CLASS EQUIVALENCE IN 8 OTTOBRE]

Definition 4.2.9: Partition

A **partition** of a set S is a set of non-empty subsets of S:

$$\{S_1, S_2, \ldots, S_n\}$$

such that every element of S is in exactly one of these subsets:

- $S_i \neq \emptyset$
- $S_i \cap S_j = \emptyset \quad \forall i \neq j$
- $\bigcup_{i=1}^{n} S_i = S$

Definition 4.2.10: Equilvance Closure

Let R be a binary relation on a set S. The **equivalence closure** R^E of R such that:

- R^E is an equivalence relation
- R^E covers R, $R \sqsubseteq R^E$
- R^E is the $\sqsubseteq$-smallest equivalence relation s.t. $R \subseteq R^E$, in other words if $\exists R' : R \subseteq R'$ and R' is an equivalence relation, then $R^E \sqsubseteq R'$.

Definition 4.2.11: Congruence Closure

Let R be a binary relation on a set S. The **congruence closure** R^C of R such that:

- R^C is a congruence relation
- R^C covers R, $R \sqsubseteq R^C$
- R^C is the $\sqsubseteq$-smallest congruence relation s.t. $R \subseteq R^C$, in other words if $\exists R' : R \subseteq R'$ and R' is a congruence relation, then $R^C \sqsubseteq R'$.

The idea is to use the congruence closure to check if a set of equalities and disequalities are satisfiable ($R = \simeq$).

If the congruence closure of the set is the equality relation put an equivality relation and a disequality relation in the same equivalence class, then the formula is unsatisfiable.

4.3 Congruence Closure Algorithm

The **Congruence Closure Algorithm** is an algorithm that computes the congruence closure of a binary relation, and it is used to check the satisfiability of a set of equalities and disequalities.

4.3.1 Idea of the CCA

Let a set of equalities and disequalities F:

$$F = \underbrace{\{s_i \simeq t_i\}_{i=1}^{n}}_{F^+} \cup \underbrace{\{s_j \not\simeq t_j\}_{j=n+1}^{n+m}}_{F^-}$$

we want to check if F is satisfiable.

We construct the set of all terms of F:

$$S_F = \{s_i \mid i = 1, \dots, n+m\} \cup \{t_i \mid i = 1, \dots, n+m\}$$

We need to find an interpretation $\mathcal{I}$ that satisfies F such that:

- $\mathcal{I} = (S_F, \Phi)$
- $\Phi(\simeq) = \sim \subseteq S_F \times S_F$ congruence relation over the terms classes
- $\begin{cases} \mathcal{I} \models s_i \simeq t_i \\ \mathcal{I} \not\models s_j \simeq t_j \end{cases} \iff \begin{cases} [s_i]_\mathcal{I} \sim [t_i]_\mathcal{I} & \forall i = 1, \dots, n \\ [s_j]_\mathcal{I} \nsim [t_j]_\mathcal{I} & \forall j = n+1, \dots, n+m \end{cases}$

So we can follow the following steps:

1. Construct the set of terms S_F.
2. Construct the initial congruence relation using the equalities in F^+ forming a sort of partiton P_F.
3. Continue to refine P_F using the equalities in F^+.
4. If a equality in P_F is found in F^-, then the formula is unsatisfiable.
5. If all the equalities in F^+ are in P_F, so that the set has only one conguence class, then the formula is satisfiable.

Example 4.3.1: Congruence Closure Algorithm - SAT

Let $F = \{x \simeq y \wedge f(x) \simeq f(z)\}$ and compute the congruence closure algorithm of F.

$$S_F = \{x, y, f(x), f(z)\}$$

We start with the initial partition:

$$P_F = \{\{x, y\}, \{f(x), f(z)\}\}$$

For the congruence axioms 3.2.2 we have:

$$x \simeq y \implies f(x) \simeq f(y)$$

So we refine the partition:

$$P_F = \{\{x, y, f(x), f(y)\}\}$$

P_F has only one equivalence class, so the formula is satisfiable.

Example 4.3.2: Congruence Closure Algorithm - UNSAT

Let $F = \underbrace{f^{(5)} \simeq x}_{EQ1} \wedge \underbrace{f^{(3)} \simeq x}_{EQ2} \wedge \underbrace{f(x) \not\simeq x}_{DEQ}.$

$$S_F = \{f^{(5)}(x), f^{(4)}(x), f^{(3)}(x), f^{(2)}(x), f(x), x\}$$

The initial partition is:

$$P_F = \{\{f^{(5)}(x)\}, \{f^{(4)}(x)\}, \{f^{(3)}(x)\}, \{f^{(2)}(x)\}, \{f(x)\}, \{x\}\}$$

We refine the partition with the equalities:

$$\begin{aligned} P_F &\overset{EQ1}{=} \{\{f^{(5)}(x), x\}, \{f^{(4)}(x)\}, \{f^{(3)}(x)\}, \{f^{(2)}(x)\}, \{f(x)\}, \{x\}\} \\ &\overset{EQ2}{=} \{\{f^{(5)}(x), f^{(3)}(x), x\}, \{f^{(4)}(x)\}, \{f^{(2)}(x)\}, \{f(x)\}, \{x\}\} \\ &\overset{EQ1 \wedge EQ2}{=} \{\{f^{(5)}(x), f^{(3)}(x), f^{(2)}(x), x\}, \{f^{(4)}(x)\}, \{f(x)\}, \{x\}\} \\ &\overset{f^{(2)}(x) \simeq x \wedge f(x) \not\simeq x}{=} \{\{f^{(5)}(x), f^{(3)}(x), f^{(2)}(x), f(x), x\}, \{f^{(4)}(x)\}\} \end{aligned}$$

Since the disequality $f(x) \not\simeq x$ is in the partition, the formula is unsatisfiable.

4.3.2 Algorithm of CCA

We can express the congruence classes as a Directed Acyclic Graph (DAG) where the nodes are the terms and the edges are the equalities.

[IMAGE OF DAG FROM 15 OTTOBRE]

We can define the data structure of the nodes of the DAG as:

Code 4.3.3: Node Stucture

```java
public class Node {
    /**
     * The unique identifier of the function or variable.
     * A costant integer.
     */
    public final Integer id;

    /**
     * The name of the function or variable.
     * A constant string.
     */
    public final String name;

    /**
     * The list of arguments associated with the function.
     * Empty for variables.
     * An ordered costant ArrayList.
     */
    private final Integer[] args;

    /**
     * The rappresentative id of the function or variable.
     * A mutable integer
     */
    public Integer find;

    /**
     * The set of congruence closure parent (ccpar).
     * An unordered mutable set of integers.
     */
    public Set<Integer> ccpar;
}
```

The Node data structure will rappresent a singular term.

Then we can define some basic functons:

Code 4.3.4: Node Function

```
Node NODE(Integer id){
    return nodes.get(id);
}
      ;
```

Code 4.3.5: Find Function

```
Integer FIND(Integer id){
    Node N = NODE(id);
    if (N.find == id){
        return id;
    } else {
        return FIND(N.id);
    }
}
      ;
```

Code 4.3.6: Ccpar Function

```
Set<Integer> CCPAR(Integer id){
    return NODE(FIND(id)).ccpar;
}
      ;
```

These functions will retrive the respective object.

We then define the Union Function wich will unite due nodes in a sigle congruence class:

Code 4.3.7: Union Function

```java
void UNION(Integer id1, Integer id2){
    Node N1 = NODE(FIND(id1));
    Node N2 = NODE(FIND(id2));
    N1.find = N2.find;
    N2.ccpar.addAll(N1.ccpar);
    N1.ccpar = Collections.emptySet();
}
        ;
```

But before the union, we need to check if the terms are congruent with each other, we define a function to do so:

Code 4.3.8: Congruent Function

```java
Boolean congruenceCheck(Node N1, Node N2){
    if (N1.name != N2.name | N1.arity != N2.arity){
        return false;
    }
    if (Arrays.equals(N1.args(), N2.args())){
        return true;
    } else {
        return false;
    }
}

Boolean CONGRUENT(Integer id1, Integer id2){
    Node N1 = NODE(id1);
    Node N2 = NODE(id2);
    return congruenceCheck(N1, N2);
}
        ;
```

We in the end define a function that procede to check if two terms of a class are congruent and union them if so, also, ccpar and rappresentative (find), are modified to respect these invariants for a **Rappresentative Node**:

- `n.id == n.fint`
- `n.ccpar` = $\varnothing$

Code 4.3.9: Merge Function

```
void MERGE(Integer id1, Integer id2){
    if (FIND(id1) != FIND(id2)) {
        Set<Integer> P1 = CCPAR(id1);
        Set<Integer> P2 = CCPAR(id2);
        UNION(id1, id2);
        for (Integer t1 : P1) {
            for (Integer t2 : P2) {
                if (FIND(t1) != FIND(t2) & CONGRUENT(
   id1, id2)){
                    MERGE(t1, t2);
                }
            }
        }
    }
}
      ;
```

4.3.3 Complexity of CCA

Let n be the number of nodes and e the number of edges in the initial DAG.

The complexity of the CCA is $O(e^2)$ for $O(n)$ calls of the merge function. Some optimizations can be done to reduce the complexity to $O(e \log e)$ for $O(n)$ calls of the merge function.

4.3.4 Optimization of CCA

We can build a non recursive version of the FIND function:

Code 4.3.10: Iterative Fine Function

```java
import java.util.Collections;

Integer ITER_FIND(Integer id){
    Node N = NODE(id);
    return N.find
}

void UNION(Integer id1, Integer id2){
    Node N1 = NODE(FIND(id1));
    Node N2 = NODE(FIND(id2));
    for (Node N : nodes.values()) {
        if (N.find == N1.find){
            N.find = N2.find;
        }
    }
    N1.find = N2.find;
    N2.ccpar.addAll(N1.ccpar);
    N1.ccpar = Collections.emptySet();
}
      ;
```

This function will find the representative of a node in a non recursive way, with the assumption that all the nodes of the non rappresentative nodes have the rappresentative node as the choosen node for the congruence class, during the union operation.

We can also apply an euristic to reduce the number of node to unite in the union operation, by choosing the node with the largest ccpar set as the representative node:

Code 4.3.11: Union Function with Euristic

```
public void EUR_UNION(int id1, int id2){
    Node N1 = NODE(FIND(id1));
    Node N2 = NODE(FIND(id2));
    Node R;
    Node N;
    if (N1.ccpar.size() > N2.ccpar.size()){
        R = N1;
        N = N2;
    } else {
        R = N2;
        N = N1;
    }
    for (Node M : nodes.values()) {
        if (M.find == N.find){
            M.find = R.find;
        }
    }
    N.find = R.find;
    R.ccpar.addAll(N1.ccpar);
    N.ccpar.clear();
}
      ;
```

In the end we can add a forbidden set to the node structure, to check forbidden merges (contraddictions), and avoid them:

Code 4.3.12: Forbidden Set Euristic

```java
private boolean FORBIDDEN(int id1, int id2){
    Node N1 = NODE(id1);
    Node N2 = NODE(id2);
    if (N1.forb.contains(N2.find) || N2.forb.contains
   (N1.find)){
        this.unsatFlag = true;
        return true;
    }
    return false;
}

public void UNION(int id1, int id2){
    Node N1 = NODE(FIND(id1));
    Node N2 = NODE(FIND(id2));
    Node R;
    Node N;
    if (N1.ccpar.size() > N2.ccpar.size()){
        R = N1;
        N = N2;
    } else {
        R = N2;
        N = N1;
    }
    N.find = R.find;
    R.ccpar.addAll(N1.ccpar);
    N.ccpar.clear();
    R.forb.addAll(N.forb);
    N.forb.clear();

}
```

In this case we can check if the merge is forbidden, and if so interrupt the entire algorithm using the `unsatFlag`.