Code Verification and Predicate Logic

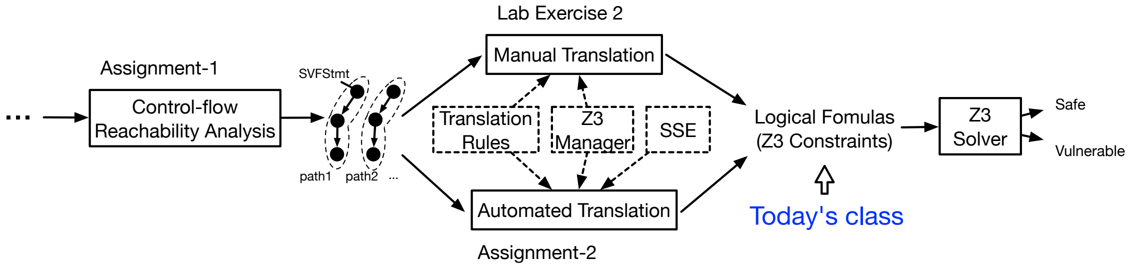
(Week 4)

Yulei Sui

School of Computer Science and Engineering

University of New South Wales, Australia

Today's class



- In **this week**, we will learn the (first-order) **logical formulas**, which are the outputs translated from code for verification.
- We will take a look at **manual and automated translation** from code to formulas for verification **from next class.**

Formal Verification For Code

$$\text{Specification} \overset{?}{\equiv} \text{code implementation}$$

Formal Verification For Code

$$\begin{tikzcd}
\text{Specification} \arrow[d, Rightarrow, "\text{translate}"] & \overset{?}{\equiv} & \text{code implementation} \arrow[d, Rightarrow, "\text{translate}"] \\
\text{logical formulas of specification} & \overset{?}{\equiv} & \text{Logical formulas of code implementation.}
\end{tikzcd}$$

Formal Verification For Code

$$\begin{tikzcd}
\text{Specification} \arrow[d, Rightarrow, "\text{translate}"] \arrow[r, phantom, "\overset{?}{\equiv}"] & \text{code implementation} \arrow[d, Rightarrow, "\text{translate}"] \\
\text{logical formulas of specification} \arrow[r, phantom, "\overset{?}{\equiv}"] & \text{Logical formulas of code implementation.}
\end{tikzcd}$$

- Proving the correctness of your code given a specification (or spec) using formal methods of mathematics
- Make the connection between specifications and implementations rigid, reliable and secure by translating specification and code into logical formulas.
- The application of theorem proving tools to perform satisfiability checking of logical formulas.

Specification[2]

- Specifications **independent of** the source code
  - Formal specification in a separate file from source code and written in a specification language and accepted by theorem provers
- Specifications **embedded in** the source code (This subject)
  - `assert` embedded in the program following the Hoare triple form.
- Hoare logic[1]: $P\ \{prog\}\ Q$.
  - P is the **pre-condition** (`assume`), expressed by predicate logic (first-order logic) formula
  - Q is the **post-condition** (`assert`)
  - *prog* is the target program
  - The Hoare triple describes that when the precondition is met, executing the program *prog* establishes the postcondition.

Assertion-Based Specification and Satisfiability

Prove whether the post-condition (`assert`) holds after executing the program given the pre-condition (`assume`).

```
assume(100 > x > 0);  // P
    foo(x){
        if(x > 10) {
            y = x + 1;
        }
        else {
            y = 10;
        }
    }
assert(y >= x + 1);  // Q
```

$$\xRightarrow{\text{translate}} \underset{\text{logical formula}}{\phi(P\{\texttt{foo}\}Q)} \xRightarrow{\text{feed into}} \text{SAT/SMT Solver}$$

Will the assertion hold?

Satisfiability Checking

Assertion verification as satisfiability checking. The assertion holds if the formula $\phi(P\{\mathtt{foo}\}Q)$ is satisfiable (SAT).

- ϕ is satisfiable if a program *prog* is correct for all valid inputs.

$$\forall \mathtt{x}\ \forall \mathtt{y}\ \ P(\mathtt{x}) \land S_{prog}(\mathtt{x}, \mathtt{y}) \rightarrow Q(\mathtt{x}, \mathtt{y})$$

- $P(x)$ is the pre-condition formula over variables $\mathtt{x}$, i.e., $\mathtt{100} > \mathtt{x} > \mathtt{0}$.
- $S_{prog}(\mathtt{x}, \mathtt{y})$ is the formula representing *prog* which accepts $\mathtt{x}$ as its input, and terminates with output $\mathtt{y}$.
- $Q(\mathtt{x}, \mathtt{y})$ is the post-condition formula over variables $\mathtt{x}, \mathtt{y}$, i.e., $\mathtt{y} >= \mathtt{x} + \mathtt{1}$.

Satisfiability Checking

Assertion verification as satisfiability checking. The assertion holds if the formula $\phi(P\{\texttt{foo}\}Q)$ is satisfiable (SAT).

- ϕ is satisfiable if a program *prog* is correct for all valid inputs.

$$\forall \mathtt{x}\ \forall \mathtt{y}\ \ P(\mathtt{x}) \land S_{prog}(\mathtt{x},\mathtt{y}) \rightarrow Q(\mathtt{x},\mathtt{y})$$

  - $P(x)$ is the pre-condition formula over variables $\mathtt{x}$, i.e., $100 > \mathtt{x} > 0$.
  - $S_{prog}(\mathtt{x},\mathtt{y})$ is the formula representing *prog* which accepts $\mathtt{x}$ as its input, and terminates with output $\mathtt{y}$.
  - $Q(\mathtt{x},\mathtt{y})$ is the post-condition formula over variables $\mathtt{x},\mathtt{y}$, i.e., $\mathtt{y} >= \mathtt{x} + 1$.
- How to prove correctness for all inputs $\mathtt{x}$? Search for counterexample $\mathtt{x}$ where ϕ does not hold, that is

$$\begin{aligned} & \exists \mathtt{x}\ \exists \mathtt{y}\ \ \neg(P(\mathtt{x}) \land S_{prog}(\mathtt{x},\mathtt{y})) \rightarrow Q(\mathtt{x},\mathtt{y})) \\ \Rightarrow\ & \exists \mathtt{x}\ \exists \mathtt{y}\ \ P(\mathtt{x}) \land S_{prog}(\mathtt{x},\mathtt{y}) \land \neg Q(\mathtt{x},\mathtt{y}) \qquad \text{(simplification}^{3}\text{)} \end{aligned}$$

  Note that $P(\mathtt{x})$ is always true if a program has no pre-condition.

Satisfiability Checking

Checking whether the logical formula ϕ is satisfiable by an SAT/SMT solver.

```
assume(100 > x > 0);
foo(x){
    if(x > 10) {
        y = x + 1;
    }
    else {
        y = 10;
    }
}
assert(y >= x + 1);
```

$$\xrightarrow{\text{translate}} \underbrace{\exists x\ \exists y\ \ P(x) \wedge S_{prog}(x, y)) \wedge \neg Q(x, y)}_{\text{logical formula } \phi} \xrightarrow{\text{feed into}} \text{SAT/SMT Solver}$$

Satisfiability Checking

Checking whether the logical formula ϕ is satisfiable by an SAT/SMT solver.

```
assume(100 > x > 0);
foo(x){
    if(x > 10) {
        y = x + 1;
    }
    else {
        y = 10;
    }
}
assert(y >= x + 1);
```

$$\xrightarrow{\text{translate}} \underbrace{\exists x\ \exists y\ \ P(x) \land S_{prog}(x, y)) \land \neg Q(x, y)}_{\text{logical formula } \phi} \xrightarrow{\text{feed into}} \text{SAT/SMT Solver}$$

Unsatisfiable! **couterexample** `x = 10` **found**!

Translating Code into Logical Formulas

- Logical formulas
  - The formulas of predicate logic are constructed from **propositional**, **predicate** and **object variables** by using **logical connectives** and **quantifiers** (This class)
- Translation
  - Translating `SVFStmts` of **each program path** (from Assignment-2) into a logical formula ϕ, and then check the satisfiablity for each path.
  - $\forall path \in prog \quad checking(\phi(path))$

    $\phi(path_1)$: $\exists \mathtt{x}\ P(\mathtt{x}) \land \big((\mathtt{x} > 10) \land (\mathtt{y} \equiv \mathtt{x} + 1)\big) \land \neg Q(\mathtt{x}, \mathtt{y})$ (`if` branch of `foo`)

    $\phi(path_2)$: $\exists \mathtt{x}\ P(\mathtt{x}) \land \big((\mathtt{x} \leq 10) \land (\mathtt{y} \equiv 10)\big) \land \neg Q(\mathtt{x}, \mathtt{y})$ (`else` branch of `foo`)

Translating Code into Logical Formulas

- Logical formulas
  - The formulas of predicate logic are constructed from **propositional**, **predicate** and **object variables** by using **logical connectives** and **quantifiers** (This class)
- Translation
  - Translating `SVFStmts` of **each program path** (from Assignment-2) into a logical formula ϕ, and then check the satisfiablity for each path.
  - $\forall path \in prog \quad checking(\phi(path))$

    $\phi(path_1)$:$\exists x(100 > x > 0) \wedge \big((\mathtt{x} > \mathtt{10}) \wedge (\mathtt{y} \equiv \mathtt{x} + \mathtt{1})\big) \wedge \neg(\mathtt{y} \geq \mathtt{x} + \mathtt{1})$ (`if` branch)

    $\phi(path_2)$:$\exists x(100 > x > 0) \wedge \big((\mathtt{x} \leq \mathtt{10}) \wedge (\mathtt{y} \equiv \mathtt{10})\big) \wedge \neg(\mathtt{y} \geq \mathtt{x} + \mathtt{1})$ (`else` branch)
  - $\phi(path_2)$: **has a counterexample** `x = 10`!!

Translating Code into Logical Formulas

- Logical formulas
  - The formulas of predicate logic are constructed from **propositional**, **predicate** and **object variables** by using **logical connectives** and **quantifiers** (This class)
- Translation
  - Translating `SVFStmts` of **each program path** (from Assignment-2) into a logical formula ϕ, and then check the satisfiablity for each path.
  - $\forall path \in prog \quad checking(\phi(path))$
    $\phi(path_1)$:$\exists x(100 > x > 0) \wedge \big((\mathtt{x} > \mathtt{10}) \wedge (\mathtt{y} \equiv \mathtt{x} + \mathtt{1})\big) \wedge \neg(\mathtt{y} \geq \mathtt{x} + \mathtt{1})$ (`if` branch)
    $\phi(path_2)$:$\exists x(100 > x > 0) \wedge \big((\mathtt{x} \leq \mathtt{10}) \wedge (\mathtt{y} \equiv \mathtt{10})\big) \wedge \neg(\mathtt{y} \geq \mathtt{x} + \mathtt{1})$ (`else` branch)
  - $\phi(path_2)$: **has a counterexample** `x = 10`!!
  - **Manual translation** via theorem prover APIs (e.g., Z3 APIs) (Assignment-3)
  - **Automatic translation** during symbolic execution (Assignment-4)

Proving each $\phi(path)$ formula

Exhaustively by not finding counterexamples via mathematical proofs.

- Direct Proof
- Proof by contradiction
- Proof by induction
- Proof by construction
- ...

Proving each $\phi(path)$ formula

Exhaustively by not finding counterexamples via mathematical proofs.

- Direct Proof
- Proof by contradiction
- Proof by induction
- Proof by construction
- ...

Manual proof is **impractical** for software verification!

- Too many variables and logical relations between them (state exploration)
- Too many assertions as specifications.

This subject **does not** focus on formal mathematical proof, but rather the **application** of **automated theorem prover** tools

- Translating code into logical formulas
- Feeding the logical formulas into a theorem prover to check the satisfiability.

Theorem Prover Tools [4]

- Interactive theorem provers (proof assistant)
  - Formal proofs by human-machine collaboration via expressive specification language and may not work directly on source code.
  - For example, ACL2, Coq, Isabelle and HOL provers
- Automated theorem provers
  - Proof automation (but less expressive than interactive provers) and can work on real-world source code.
  - For example, Z3 and CVC

Automated Theorem Provers

A prover/solver checks if a formula $\phi(P\{\texttt{foo}\}Q)$ is satisfiable (SAT).

- If yes, the solver returns a **model** *m*, a valuation of $\texttt{x}, \texttt{y}, \texttt{z}$ of `foo` that satisfies ϕ (i.e., *m* makes ϕ true).
- Otherwise, the solver returns unsatisfiable (UNSAT)

SAT vs. **SMT** solvers

- **SAT** solvers accept **propositional logic** (Boolean) formulas, typically in the conjunctive normal form (CNF).
- **SMT** (satisfiability modulo theories) solvers generalize the Boolean satisfiability problem (SAT), and accept more expressive **predicate logic** formulas, i.e., propositional logic with predicates and quantification.
    - Z3 Automated Theorem Prover[5], a cross-platform satisfiability modulo theories (SMT) solver developed by Microsoft (This subject).
    - More details next week..

Predicate Logic in Code Verification

(Week 4)

Yulei Sui

School of Computer Science and Engineering

University of New South Wales, Australia

Propositional and Predicate Logic

The following will be the background to understand some terms when using a theorem prover (SAT/SMT solver).

If you have already learned basic discrete math and logic theory, you can skip this. :)

Discrete Math and Basic Logic Theory

Strongly suggest you revisit or pick up discrete math if you haven't. You can learn through the following learning materials or search google for more materials:

- Discrete mathematics wiki
  `https://en.wikipedia.org/wiki/Discrete_mathematics`
- Discrete math videos:
  `https://www.youtube.com/hashtag/discretemathematicsbyneso`
- Propositional logic
  `https://en.wikipedia.org/wiki/Propositional_calculus`
- Predicate logic (or first-order logic)
  `https://en.wikipedia.org/wiki/First-order_logic`
- Discrete mathematics book
  `http://discrete.openmathbooks.org/dmoi3.html`

Satisfiability Solving as Logic Inference Problem

Logic Inference Problem:

- Given:
    - A knowledge base *KB* (a set of constraints (logical formulas) extracted from code statements) and an assertion *Q*, For example,

Satisfiability Solving as Logic Inference Problem

Logic Inference Problem:

- Given:
  - A knowledge base *KB* (a set of constraints (logical formulas) extracted from code statements) and an assertion *Q*, For example,
  - *KB* : $((\mathrm{x} > \mathrm{z}) \wedge (\mathrm{y} == \mathrm{x} + 1)) \vee ((\mathrm{x} \leq \mathrm{z}) \wedge (\mathrm{y} == 10))$
  - *Q* : $\mathrm{y} \geq \mathrm{x} + 1$
- $KB \vdash Q$?
  - Does KB semantically entail *Q*?
  - If all constraints in *KB* are true, is the assertion true?
  - Is the specification *Q* satisfiable given constraints from code?
- Each element (**proposition** or **predicate**) in *KB* can be seen as a **premise** and *Q* is the **conclusion.**

Propositional Logic (Statement Logic) [6]

A **proposition** is a statement that is either true or false. Propositional logic studies the ways statements can interact with each other.

- **Propositional variables** (e.g., P and Q) represent propositions or statements in the formal system.
- A **propositional formula** is logical formula with **propositional variables** and **logical connectives** like `and` ($\wedge$) , `or` ($\vee$), `negation` ($\neg$), `implication` ($\Rightarrow$)
- **Inference rules** allow certain formulas to be derived. These derived formulas are called **theorems** (or true propositions). The derivation can be interpreted as proof of the proposition represented by the theorem.

Propositional Logic (Natural Language Example)

- A natural language inference example where **both premises and conclusion are propositions** in the form of natural language statements.

Premise 1	: If you get 85 marks, then you get a high distinction.
Premise 2	: You get 85 marks.
Conclusion	: You get a high distinction.

Propositional Logic (Natural Language Example)

- A natural language inference example where **both premises and conclusion are propositions** in the form of natural language statements.

Premise 1	: If you get 85 marks, then you get a high distinction.
Premise 2	: You get 85 marks.
Conclusion	: You get a high distinction.

- Propositional variable representation of the above inference rule (P is interpreted as “you get 85 marks“ and Q is “you get a high distinction“)

Premise 1	$P \to Q$
Premise 2	P
Conclusion	Q

- The inference rule: for any P and Q, whenever $P \to Q$ and P are true, necessarily Q is true, written in Modus ponens form: $\{P, P \to Q\} \vdash Q$

Propositional Logic (Code Example)

- A code example where **both premises and conclusion are propositions** in this inference rule.

Premise 1	: if(x $>$ 10 && y $<$ 5) z = 15;
Premise 2	: x $>$ 10;
Premise 3	: y $<$ 5;
Conclusion	: z = 15;

Propositional Logic (Code Example)

- A code example where **both premises and conclusion are propositions** in this inference rule.

Premise 1	: if(x $>$ 10 && y $<$ 5) z = 15;
Premise 2	: x $>$ 10;
Premise 3	: y $<$ 5;
Conclusion	: z = 15;

- Propositional variable representation of the above inference rule (P_1 is interpreted as “x $>$ 10“, P_2 is “y $<$ 5“ and Q is “z = 15“)

Premise 1	: $(P_1 \wedge P_2) \rightarrow Q$
Premise 2	: P_1
Premise 3	: P_2
Conclusion	: Q

- Succinctly written as: $\{P_1, P_2, (P_1 \wedge P_2) \rightarrow Q\} \vdash Q$

Limitations of Propositional Logic (Natural Language Example)

The following valid argument **can** be inferred using propositional logic, i.e., $\{P, P \to Q\} \vdash Q$

- $P \to Q$: If you get 85 marks, then you get a high distinction.
- P : You get 85 marks.
- Q : You get a high distinction.

The following valid argument **can not** be inferred using propositional logic, i.e., $\{P', P \to Q\} \nvdash Q'$ since propositions P' and P are not interpreted as the same.

- $P \to Q$: If you get 85 marks, then you get a high distinction.
- P' : Jack gets 85 marks.
- Q' : Jack gets a high distinction.

Propositional Logic is **less expressive** and has **weak generalization power**. It does not allow us to conclude the truth of ALL or SOME statements. It is not possible to mention properties of objects in the statement, or relationships between properties.

Limitations of Propositional Logic (Code Example)

The following valid argument **can** be inferred using propositional logic, i.e.,$\{P_1, P_2, (P_1 \land P_2) \rightarrow Q\} \vdash Q$

- P_1: x $>$ 10;
- P_2: y $<$ x;
- $(P_1 \land P_2) \rightarrow Q$: if(x $>$ 10 && y $<$ x) z = 15;
- Q: z = 15;

The following valid argument **can not** be inferred using propositional logic, i.e., $\{P'_1, P'_2, (P_1 \land P_2) \rightarrow Q\} \nvdash Q$

- P'_1: x $=$ 11;
- P'_2: y $=$ 10;
- $(P_1 \land P_2) \rightarrow Q$: if(x $>$ 10 && y $<$ x) z = 15;
- Q: z = 15;

Predicate Logic (First-Order Logic) [7]

First-order logic is propositional logic with predicates and quantification.

- Propositional logic: boolean logic which represents statements without reflecting their structures and relations
- Predicate logic: is more expressive and further analyzes proposition(s) by representing their entities' properties and relations and to group entities, i.e., additionally covers predicates and quantification.

Predicate Logic (First-Order Logic) [7]

First-order logic is propositional logic with predicates and quantification.

- Propositional logic: boolean logic which represents statements without reflecting their structures and relations
- Predicate logic: is more expressive and further analyzes proposition(s) by representing their entities' properties and relations and to group entities, i.e., additionally covers predicates and quantification.
- A **predicate** R takes one or more variables/entities as input and outputs a proposition and has a truth value (either true or false).
  - A statement whose truth value is dependent on variables.
  - For example, in $R(x) : x > 5$, "x" is the variable and "> 5" is the predicate. After assigning x with the value 6, $R(x)$ becomes a proposition $6 > 5$.
- A **quantifier** is applied to a set of entities
  - Universal quantifier $\forall$, meaning all, every
  - Existential quantifier $\exists$, meaning some, there exists

Predicate Logic (Natural Language Example)

Consider the two statements

- “Jack got a high distinction”
- “Peter got a high distinction”

In propositional logic, these statements are viewed as being unrelated and the sub-statements/words/entities are not further analyzed.

- **Predicate logic** allows us to define a **predicate** R representing “got a high distinction” which occurs in both sentences.
- $R(x)$ is the **predicate logic statement (formula)** which accepts a name x and output as “x got a high distinction”.

Predicate Logic (Code Example)

- P_1: x $>$ 10;
- P_2: y $<$ x;
- $(P_1 \wedge P_2) \rightarrow Q$: if(x $>$ 10 && y $<$ x) z = 15;
- Q: z = 15;

In propositional logic, each code statement (including its variables) is viewed as one proposition. Its variables and their relations are not further analyzed.

- **Predicate logic** allows us to define the following three predicates
  - $R_1(x)$ representing x $>$ 10 for the property of a single variable.
  - $R_2(y, x)$ representing y $<$ x for the relation between two variables.
  - $R_3(z)$ representing z = 15 for the property of a single variable.

Predicate Logic (Code Example)

- P_1: x $>$ 10;
- P_2: y $<$ x;
- $(P_1 \wedge P_2) \rightarrow Q$: if(x $>$ 10 && y $<$ x) z = 15;
- Q: z = 15;

In propositional logic, each code statement (including its variables) is viewed as one proposition. Its variables and their relations are not further analyzed.

- **Predicate logic** allows us to define the following three predicates
  - $R_1(x)$ representing x $>$ 10 for the property of a single variable.
  - $R_2(y, x)$ representing y $<$ x for the relation between two variables.
  - $R_3(z)$ representing z = 15 for the property of a single variable.

Given **a set of propositions/predicates** as the knowledge base *KB*, let us take a look at how we do **the inference** $KB \vdash Q$ using propositional logic and predicate logic. Does *KB* semantically entail *Q*?

Predicate Logic (Code Example)

- P_1: x $>$ 10;
- P_2: y $<$ x;
- $(P_1 \wedge P_2) \rightarrow Q$: if(x $>$ 10 && y $<$ x) z = 15;
- Q: z = 15;

In propositional logic, each code statement (including its variables) is viewed as one proposition and its variables are not further analyzed.

- **Predicate logic** allows us to define the following three predicates
  - $R_1(x)$ representing x $>$ 10 for relation/property of a single variable.
  - $R_2(y, x)$ representing y $<$ x for relation between two variables.
  - $R_3(z)$ representing z = 15 for relation/property of a single variable.

Predicate Logic (Code Example)

- P_1: x $>$ 10;
- P_2: y $<$ x;
- $(P_1 \wedge P_2) \rightarrow Q$: if(x $>$ 10 && y $<$ x) z = 15;
- Q: z = 15;

In propositional logic, each code statement (including its variables) is viewed as one proposition and its variables are not further analyzed.

- **Predicate logic** allows us to define the following three predicates
  - $R_1(x)$ representing x $>$ 10 for relation/property of a single variable.
  - $R_2(y, x)$ representing y $<$ x for relation between two variables.
  - $R_3(z)$ representing z = 15 for relation/property of a single variable.

Propositional logical for the inference

- $\{P_1, P_2, (P_1 \wedge P_2) \rightarrow Q\} \vdash Q$

Predicate logical for the inference

- $\{R_1(x), R_2(y, x), (R_1(x) \wedge R_2(y, x)) \rightarrow R_3(z)\} \vdash Q$

Predicate Logic (Code Example)

- P_1: x $>$ 10;
- P_2: y $<$ x;
- $(P_1 \wedge P_2) \rightarrow Q$: if(x $>$ 10 && y $<$ x) z = 15;
- Q: z = 15;

In propositional logic, each code statement (including its variables) is viewed as one proposition and its variables are not further analyzed.

- **Predicate logic** allows us to define the following three predicates
  - $R_1(x)$ representing x $>$ 10 for relation/property of a single variable.
  - $R_2(y, x)$ representing y $<$ x for relation between two variables.
  - $R_3(z)$ representing z = 15 for relation/property of a single variable.

Propositional logical for the inference

- $\{P_1, P_2, (P_1 \wedge P_2) \rightarrow Q\} \vdash Q$

Predicate logical for the inference

- $\{R_1(x), R_2(y, x), (R_1(x) \wedge R_2(y, x)) \rightarrow R_3(z)\} \vdash Q$
- $\{x > 10, y < x, (x > 10 \wedge y < x) \rightarrow z = 15\} \vdash z = 15$

Verification as Solving Constrained Horn Clauses

A constrained horn clause has the following form:

- $\forall V \ (\varphi \wedge R_1(X_1) \wedge \ldots \wedge R_k(X_k)) \rightarrow Q(X)$
    - τ is a **background theory** (e.g., Linear Arithmetic, Arrays, BitVectors, or combinations of the above)
    - V are variables, and X_i is a set of **terms** over V
    - φ is a constraint in the background theory τ
    - $R_1, \ldots R_k$ and Q are multi-ary **predicates**.
    - $R_i(X_i)$ is an application of predicate R_i to **first-order terms**.

Verification as solving constrained horn clauses.