Lab: Z3 Theorem Prover

(Week 4)

Yulei Sui

School of Computer Science and Engineering

University of New South Wales, Australia

Quiz-1 and Exercise-1 Marks Released

Marks are out and let us go through Quiz-1 questions!

Quiz-2 and Exercise-2

Remember to `git pull` or `docker pull`!

You **MUST** update your code template to the latest version for Lab-Exercise-2. Otherwise, you will not receive marks because Lab-2 and Assignment-2 have been changed to differentiate from previous years.

Quiz-2 and Exercise-2

- Quiz-2 with 25 questions (5 points), **due date: 23:59 Tuesday, Week 7**
  - Logical formula and predicate logic
  - Z3's knowledge and translation rules
- Lab-Exercise-2 (5 points), **due date: 23:59 Tuesday, Week 7**
  - **Goal:** Manually translate code into z3 formulas/constraints and verify the assertions embedded in the code.
  - **Specification:** `https://github.com/SVF-tools/Software-Security-Analysis/wiki/Lab-Exercise-2`
  - **SVF Z3 APIs:** `https://github.com/SVF-tools/Software-Security-Analysis/wiki/SVF-Z3-API`
- Assignment-2 (25 points) will **start from Week 5 and due date: 23:59 Tuesday, Week 8**
  - **Goal:** automatically perform assertion-based verification for code using static symbolic execution.
  - **Specification:** `https://github.com/SVF-tools/Software-Security-Analysis/wiki/Assignment-2`

Intraprocedural Example

```
1 int* p;
2 int q;
3 int* r;
4 int x;
5 p = malloc(...);
6 q = 5;
7 *p = q;
8 x = *p;
9 assert(x==10);
```

Source code

```
1 expr p = getZ3Expr("p");
2 expr q = getZ3Expr("q");
3 expr r = getZ3Expr("r");
4 expr x = getZ3Expr("x");
5 printExprValues();
```

Translation code using Z3Mgr

```
----------Var and Value-----


----------------------------
```

nothing printed because expressions have no value

Output on terminal

Intraprocedural Example

```
1 int* p;
2 int q;
3 int* r;
4 int x;
5 p = malloc(...);
6 q = 5;
7 *p = q;
8 x = *p;
9 assert(x==10);
```

Source code

```
1 expr p = getZ3Expr("p");
2 expr q = getZ3Expr("q");
3 expr r = getZ3Expr("r");
4 expr x = getZ3Expr("x");
5 expr malloc1 = getMemObjAddress("malloc1");
6 addToSolver(p == malloc1);
7 printExprValues();
```

Translation code using Z3Mgr

```
----------Var and Value-----
Var5 (malloc1)    Value: 0x7f000005
Var1 (p)          Value: 0x7f000005
----------------------------
```

`0x7f000005` (or 2130706437 in decimal) represents the virtual memory address of this object

Each `ObjVar` starts with `0x7f` + its ID.

Output on terminal

Intraprocedural Example

```
1 int* p;
2 int q;
3 int* r;
4 int x;
5 p = malloc(...);
6 q = 5;
7 *p = q;
8 x = *p;
9 assert(x==10);
```

Source code

```
1  expr p = getZ3Expr("p");
2  expr q = getZ3Expr("q");
3  expr r = getZ3Expr("r");
4  expr x = getZ3Expr("x");
5  expr malloc1 = getMemObjAddress("malloc1");
6  addToSolver(p == malloc1);
7  addToSolver(q == getZ3Expr(5));
8  storeValue(p, q);
9  addToSolver(x == loadValue(p));
10 printExprValues();
```

Translation code using Z3Mgr

```
----------Var and Value-----
Var5 (malloc1)    Value: 0x7f000005
Var1 (p)          Value: 0x7f000005
Var2 (q)          Value: 5
Var4 (x)          Value: 5
----------------------------
```

store value of `q` to address `0x7f000005`

load the value from `0x7f000005` to `x`

Output on terminal

Intraprocedural Example

```
1 int* p;
2 int q;
3 int* r;
4 int x;
5 p = malloc(...);
6 q = 5;
7 *p = q;
8 x = *p;
9 assert(x==10);
```

```
1  expr p = getZ3Expr("p");
2  expr q = getZ3Expr("q");
3  expr r = getZ3Expr("r");
4  expr x = getZ3Expr("x");
5  expr malloc1 = getMemObjAddress("malloc1");
6  addToSolver(p == malloc1);
7  addToSolver(q == getZ3Expr(5));
8  storeValue(p, q);
9  addToSolver(x == loadValue(p));
10 printExprValues();
11
12 // use checkNegateAssert to unsat of x!=10
13 // could check sat of x==10 due to
14 // closed-world program
```

```
----------Var and Value-----
Var5 (malloc1)    Value: 0x7f000005
Var1 (p)          Value: 0x7f000005
Var2 (q)          Value: 5
Var4 (x)          Value: 5

Assertion failed: (false &&
"The assertion is unsatisfiable");
---------------------------
```

Contradictory Z3 constraints!

$x \equiv 5$ contradicts $x \equiv 10$

Source code

Translation code using Z3Mgr

Output on terminal

Intraprocedural Example

```
1 int* p;
2 int q;
3 int* r;
4 int x;
5 p = malloc(...);
6 q = 5;
7 *p = q;
8 x = *p;
9 assert(x==10);
```

```
1  expr p = getZ3Expr("p");
2  expr q = getZ3Expr("q");
3  expr r = getZ3Expr("r");
4  expr x = getZ3Expr("x");
5  expr malloc1 = getMemObjAddress("malloc1");
6  addToSolver(p == malloc1);
7  addToSolver(q == getZ3Expr(5));
8  storeValue(p, q);
9  addToSolver(x == loadValue(p));
10 printExprValues();
11
12 /// evaluation code as below
13 std::cout<< getEvalExpr(x == getZ3Expr(10))
14 << std::endl;
```

```
----------Var and Value-----
Var5 (malloc1)    Value: 0x7f000005
Var1 (p)          Value: 0x7f000005
Var2 (q)          Value: 5
Var4 (x)          Value: 5
false
----------------------------
```

There is no model available (unsat)

when evaluating `x == getZ3Expr(10)`

Source code

Translation code using Z3Mgr

Output on terminal

Interprocedural Example (Call and Return)

```
1  int bar(int a){
2      int r = a;
3      return r;
4  }
5  void main(){
6      int p, q;
7      p = bar(2);
8      q = bar(3);
9      assert(p==2)
10 }
```

Source code

```
1  expr p = getZ3Expr("p");
2  expr q = getZ3Expr("q");
3  solver.push();
4  expr a = getZ3Expr("a");
5  addToSolver(a == getZ3Expr(2));
6  solver.check();
7  expr r = getEvalExpr(a);
8  printExprValues();
9  solver.pop();
10 addToSolver(p == r);
```

Handle first callsite `p=bar(2)`

Translation code using Z3Mgr

```
----------Var and Value-----
Var2 (a)          Value: 2
-----------------------------
```

(1) `push` the z3 constraints when calling `bar` and `pop` when returning from `bar`

(2) Expression `r` is the return value evaluated from `a` after returning from callee `bar`

Output on terminal

Interprocedural Example (Call and Return)

```
1  int bar(int a){
2      int r = a;
3      return r;
4  }
5  void main(){
6      int p, q;
7      p = bar(2);
8      q = bar(3);
9      assert(p==2)
10 }
```

```
1  expr p = getZ3Expr("p");
2  expr q = getZ3Expr("q");
3  solver.push();
4  expr a = getZ3Expr("a");
5  addToSolver(a == getZ3Expr(2));
6  solver.check();
7  expr r = getEvalExpr(a);
8  solver.pop();
9  addToSolver(p == r);
10 printExprValues();
```

Handle first callsite `p=bar(2)`

```
----------Var and Value-----
Var1 (p)        Value: 2
----------------------------
```

Now we only have `p`'s value and `a` is not in the current stack since constraint `a == getZ3Expr(2)` has been popped

Source code

Translation code using Z3Mgr

Output on terminal

Interprocedural Example (Call and Return)

```
1  int bar(int a){
2      int r = a;
3      return r;
4  }
5  void main(){
6      int p, q;
7      p = bar(2);
8      q = bar(3);
9      assert(p==2)
10 }
```

```
1  expr p = getZ3Expr("p");
2  expr q = getZ3Expr("q");
3  solver.push();
4  expr a = getZ3Expr("a");
5  addToSolver(a == getZ3Expr(2));
6  expr r = getEvalExpr(a);
7  solver.pop();
8  addToSolver(p == r);
9  solver.push();
10 addToSolver(a == getZ3Expr(3));
11 r = getEvalExpr(a);
12 solver.pop();
13 addToSolver(q == r);
14 printExprValues();
```

Handle second callsite `q=bar(3)`

```
----------Var and Value-----
Var1 (p)         Value: 2
Var2 (q)         Value: 3
----------------------------
```

We have two expressions and their values in main's scope

Source code | Translation code using Z3Mgr | Output on terminal

Bad Interprocedural Example Without `push/pop`

```
1  int bar(int a){
2      int r = a;
3      return r;
4  }
5  void main(){
6      int p, q;
7      p = bar(2);
8      q = bar(3);
9      assert(p==2)
10 }
```

Source code

```
1  expr p = getZ3Expr("p");
2  expr q = getZ3Expr("q");
3  expr a = getZ3Expr("a");
4  addToSolver(a == getZ3Expr(2));
5  expr r = getEvalExpr(a);
6  addToSolver(p == r);
7  addToSolver(a == getZ3Expr(3));
8  r = getEvalExpr(a);
9  addToSolver(q == r);
10 printExprValues();
```

Translation code using Z3Mgr

```
----------Var and Value-----
Assertion failed: (res!=z3::unsat &&
"unsatisfied constraints! Check your
contradictory constraints added to
the solver")
----------------------------
```

Output on terminal

both `a == getZ3Expr(2)` and `a == getZ3Expr(3)` are added into the solver in the same scope

Bad Interprocedural Example Without Evaluating Return

```
1  int bar(int a){
2      int r = a;
3      return r;
4  }
5  void main(){
6      int p, q;
7      p = bar(2);
8      q = bar(3);
9      assert(p==2)
10 }
```

Source code

```
1  expr p = getZ3Expr("p");
2  expr q = getZ3Expr("q");
3  expr r = getZ3Expr("r");
4  expr a = getZ3Expr("a");
5  solver.push();
6  addToSolver(a == getZ3Expr(2));
7  addToSolver(r == a);  // invalid after pop
8  solver.pop();
9  addToSolver(p == r);
10 printExprValues();
11 solver.push();
12 addToSolver(a == getZ3Expr(3));
13 addToSolver(r == a);  // invalid after pop
14 solver.pop();
15 addToSolver(q == r);
16 printExprValues();
```

Translation code using Z3Mgr

```
----------Var and Value-----
Var1 (p)         Value: random
Var2 (q)         Value: random
Var3 (r)         Value: random
----------------------------
```

Output on terminal

the values of `p,q,r` are

the same random number

Array and Struct Example

```
1  void main(){
2      int* a;
3      int* x;
4      int y;
5      a = malloc(...);
6      x = &a[2];
7      *x = 3;
8      y = *x;
9      assert(y==3);
10 }
```

Source code

```
1  expr a = getZ3Expr("a");
2  expr x = getZ3Expr("x");
3  expr y = getZ3Expr("y");
4  addToSolver(a == getMemObjAddress("malloc"));
5  addToSolver(x == getGepObjAddress(a,2));
6  storeValue(x, getZ3Expr(3));
7  addToSolver(y == loadValue(x));
8
9  /// print expr values as below
10 printExprValues();
```

Translation code using Z3Mgr

```
----------Var and Value-----
Var1 (a)          Value: 0x7f000004
Var4 (malloc)     Value: 0x7f000004
Var2 (x)          Value: 0x7f000003
Var3 (y)          Value: 0x7f000003
----------------------------
```

Output on terminal

`getGepObjAddress` returns the field address of the aggregate object *a*

The virual address also in the form of `0x7f.. + VarID`

Array and Struct Example

```
1  void main(){
2      int* a;
3      int* x;
4      int y;
5      a = malloc(...);
6  // similar for struct
7  //  x=&(a->fld2)
8      x = &a[2];
9      *x = 3;
10     y = *x;
11     assert(y==3);
12 }
```

Source code

```
1  expr a = getZ3Expr("a");
2  expr x = getZ3Expr("x");
3  expr y = getZ3Expr("y");
4  addToSolver(a == getMemObjAddress("malloc"));
5  addToSolver(x == getGepObjAddress(a,2));
6  storeValue(x, getZ3Expr(3));
7  addToSolver(y == loadValue(x));
8
9  /// print expr values as below
10 printExprValues();
```

Translation code using Z3Mgr

```
----------Var and Value-----
Var1 (a)          Value: 0x7f000004
Var4 (malloc)     Value: 0x7f000004
Var2 (x)          Value: 0x7f0001f7
Var3 (y)          Value: 3
----------------------------
```

`getEvalExpr` retrieve the value from the expression

Output on terminal

Branch Example

```
1  void main(int argv){
2      int y = 2;
3      if(argv > 2)
4          y = argv;
5
6  // both definitions of y
7  // at lines 2 and 4 go to
8  // this joint point
9      assert(y>=2);
10 }
```

Source code

```
1  expr argv = getZ3Expr("argv");
2  expr y = getZ3Expr("y");
3  // new variable y1 to mimic the phi to merge
4  // two definitions of y at control flow joint point
5  expr y1 = getZ3Expr("y1");
6  expr two = getZ3Expr(2);
7  addToSolver(y == two);
8  addToSolver(y1 == ite(argv > two, argv, y));
9
10 /// expr validation and evaluation as below
11 printExprValues();
12 std::cout<<getEvalExpr(y1 >= two)<<"\n";
```

Translation code using Z3Mgr

```
----------Var and Value-----
Var1 (argv)          Value: 0
Var2 (y)             Value: 2
Var3 (y1)            Value: 2
----------------------------
true
```

Output on terminal

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* : $((\mathtt{x} > \mathtt{z}) \wedge (\mathtt{y} == \mathtt{x} + 1)) \vee ((\mathtt{x} \leq \mathtt{z}) \wedge (\mathtt{y} == 10))$
  - *Q* : $\mathtt{y} \geq \mathtt{x} + 1$
- $KB \vdash Q$?
  - 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) [1]

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., S) 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 (S is interpreted as “you get 85 marks“ and Q is “you get a high distinction“)

  Premise 1 $S \rightarrow Q$
  Premise 2 S
  Conclusion Q

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

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 (S_1 is interpreted as "x $>$ 10", S_2 is "y $<$ 5" and Q is "z = 15")

  Premise 1 : $(S_1 \wedge S_2) \rightarrow Q$
  Premise 2 : S_1
  Premise 3 : S_2
  Conclusion : Q

- Succinctly written as: $\{S_1, S_2, (S_1 \wedge S_2) \rightarrow Q\} \vdash Q$

Limitations of Propositional Logic (Natural Language Example)

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

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

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

- $S \rightarrow Q$: If you get 85 marks, then you get a high distinction.
- S' : 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.,$\{S_1, S_2, (S_1 \wedge S_2) \rightarrow Q\} \vdash Q$

- S_1: x $>$ 10;
- S_2: y $<$ x;
- $(S_1 \wedge S_2) \rightarrow Q$: if(x $>$ 10 && y $<$ x) z = 15;
- Q: z = 15;

The following valid argument **can not** be inferred using propositional logic, i.e., $\{S_1', S_2', (S_1 \wedge S_2) \rightarrow Q\} \nvdash Q$

- S_1': x $=$ 11;
- S_2': y $=$ 10;
- $(S_1 \wedge S_2) \rightarrow Q$: if(x $>$ 10 && y $<$ x) z = 15;
- Q: z = 15;

Predicate Logic (First-Order Logic) [2]

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) [2]

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** P 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 $P(x) : x > 5$, "x" is the variable and "> 5" is the predicate. After assigning x with the value 6, $P(x)$ 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** P representing “got a high distinction” which occurs in both sentences.
- $P(x)$ is the **predicate logic statement (formula)** which accepts a name x and output as “x got a high distinction”.

Predicate Logic (Code Example)

- S_1: x $>$ 10;
- S_2: y $<$ x;
- $(S_1 \wedge S_2) \rightarrow Q$: 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
  - $P_1(x)$ representing x $>$ 10 for the property of a single variable.
  - $P_2(y, x)$ representing y $<$ x for the relation between two variables.
  - $P_3(z)$ representing z = 15 for the property of a single variable.

Predicate Logic (Code Example)

- S_1: x $>$ 10;
- S_2: y $<$ x;
- $(S_1 \wedge S_2) \rightarrow Q$: 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
  - $P_1(x)$ representing x $>$ 10 for the property of a single variable.
  - $P_2(y, x)$ representing y $<$ x for the relation between two variables.
  - $P_3(z)$ 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)

- S_1: x $>$ 10;
- S_2: y $<$ x;
- $(S_1 \wedge S_2) \rightarrow Q$: 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
  - $P_1(x)$ representing x $>$ 10 for relation/property of a single variable.
  - $P_2(y, x)$ representing y $<$ x for relation between two variables.
  - $P_3(z)$ representing z = 15 for relation/property of a single variable.

Predicate Logic (Code Example)

- S_1: x $>$ 10;
- S_2: y $<$ x;
- $(S_1 \wedge S_2) \rightarrow Q$: 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
  - $P_1(x)$ representing x $>$ 10 for relation/property of a single variable.
  - $P_2(y, x)$ representing y $<$ x for relation between two variables.
  - $P_3(z)$ representing z = 15 for relation/property of a single variable.

Propositional logical for the inference

- $\{S_1, S_2, (S_1 \wedge S_2) \rightarrow Q\} \vdash Q$

Predicate logical for the inference

- $\{P_1(x), P_2(y, x), (P_1(x) \wedge P_2(y, x)) \rightarrow P_3(z)\} \vdash Q$

Predicate Logic (Code Example)

- S_1: x $>$ 10;
- S_2: y $<$ x;
- $(S_1 \wedge S_2) \rightarrow Q$: 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
  - $P_1(x)$ representing x $>$ 10 for relation/property of a single variable.
  - $P_2(y, x)$ representing y $<$ x for relation between two variables.
  - $P_3(z)$ representing z = 15 for relation/property of a single variable.

Propositional logical for the inference

- $\{S_1, S_2, (S_1 \wedge S_2) \rightarrow Q\} \vdash Q$

Predicate logical for the inference

- $\{P_1(x), P_2(y, x), (P_1(x) \wedge P_2(y, x)) \rightarrow P_3(z)\} \vdash Q$
- $\{\text{x} > 10, \text{y} < \text{x}, (\text{x} > 10 \wedge \text{y} < \text{x}) \rightarrow \text{z} = 15\} \vdash \text{z} = 15$

Verification as Solving Constrained Horn Clauses

A constrained horn clause has the following form:

- $\forall V\ (\varphi \wedge P_1(X_1) \wedge ... \wedge P_k(X_k)) \rightarrow Q(X)$
  - 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
  - $P_1, ...P_k$ and Q are multi-ary **predicates**.
  - $P_i(X_i)$ is an application of predicate P_i to **first-order terms**.

Verification as solving constrained horn clauses.