Residue Generator

Introduction

This tool implements the partial subsumption algorithm introduced in [1]. The partial subsumption algorithm extends the basic subsumption algorithm[2] by finding a substitution of a clause (the subsuming one) that matches it to another clause (the one being subsumed) as much as possible, and get the fragment of the subsuming clause that cannot match with the subsumed clause. Such fragment is called "residue".

(Please refer to the readme.pdf for proper rendering of mathematical formulas.)

Definition: A Clause C subsumes a clause D if and only if there is a substitution σ such that $C\sigma \vDash D$. D is called a subsumed clause[1].

Example: Let $C = P(x)$ and $D = P(a) \vee Q(a)$. If $\sigma = \{a/x\}$, then $C\sigma = P(a)$. Since $C\sigma \subseteq D$, C subsumes D.

Definition: A clause C partially subsumes a clause D if and only if C does not subsumes D but a non-null subclause of C subsumes D.

Example: Let $C = P(x) \vee R(x)$ and $D = P(a) \vee Q(a)$. While C does not subsumes D, its subclause $P(x)$ subsumes D with $\sigma = \{a/x\}$, thus C partially subsumes D.

Definition: If a clause C partially subsumes a clause D with a subclause C' under substitution σ, then $R = (C - C')\sigma$ is the residue of C w.r.t. D.

Example: Let $C = P(x) \vee Q(x)$ and $D = P(a) \vee Q(b)$ where x is a variable and a, b are two different constants. Since $P(x)$ subsumes D under substitution $\{a/x\}$ and $Q(x)$ subsumes D under substitution $\{b/x\}$, $Q(a)$ and $P(b)$ are two residues of C w.r.t. D.

Usage Guide

Prerequisites

This tool is implemented in Python 3 and invokes the Z3 Python API in the background. Please have Python 3 and the Z3 installed in your system, and make sure the Z3 Python module could be properly imported in your Python environment.

- Python3
- The Z3 theorem prover

Basic Classes

The implementation of this tool is object oriented. This section introduces some basic classes that users will interact with when using this tool.

Rule Class

The Rule class is designed to hold a datalog rule which is a horn clause expressed in implication form:

$$h \leftarrow b1, b2, ..., bn$$

Where h is the head of the rule, and $\leftarrow b1, b2, ..., bn$ is the body. It means *if* $b1 \wedge b2 \wedge ... \wedge bn$ *is true, then* h *must also be true.* h and bi are literals that implemented using the `Literal` class.

The partial subsumption algorithm is implemented as a method of the `Rule` class. It should be called from the subsuming clause and the subsumed clause is passed to the method as its single argument. The residues are returned as a list of `Rule` objects. If there is no resolution between the two clauses, then the returned list contains only the maximal residue which is the subsuming clause itself. The following codes finds the residues of the `subsumingClause` w.r.t. `subsumedClause` and print the results:

```
# literal1, literal2 and literal3 are instances of the Literal class
subsumingClause = Rule(body = [literal1, literal2])
print(f"subsuming clause:\n{subsumingClause.show()}")

subsumedClause = Rule(body = [literal1, literal2, literal3])
print(f"subsumed clause:\n{subsumedClause.show()}")

R = subsumingClause.partial_subsume(subsumedClause)
print("residues:")
for r in R:
    print(r.show())
```

Note: even though the Rule class can be used to store any horn clause which may or may not have a head, the partial subsumption algorithm, however, only handles goal clauses that have no head.

Literal Class

The Literal class is the base class for literals. We classify the literals in two categories: nonevaluable literals and evaluable literals. A nonevaluable literal is a formula declared by user that this tool has not knowledge to evaluate if it's True or not, while an evaluable literal is a comparison operator that can be evaluated by this tool. Currently, the tool supports the following equality and inequality operators: **Equal** (==), **Not equal** (!=), **Greater** (>), **Greater or equal** (>=), **Smaller** (<) and **Smaller or equal** (<=).

To create an nonevaluable literal, instantiate the `Literal` class with the name of the literal and its arguments. For example, the following codes create a negated literal named **R** that have 3 arguments (the 3 arguments are passed to the `Literal` class constructor in a list, and arguments are instances of the `MySymbol` class):

```
# the keyword argument negated is False by default
p = Literal('R', [sym1, sym2, sym3], negated = True)
```

To create an evaluable literal, simply write `lhs operator rhs` , where `lhs` and `rhs` are the left and the right hand side symbol respectively, and the `operator` is one of the supported comparison operators (the comparison operators are overridden for the `MySymbol` class and they will return an instance of `ComparisonLiteral` which is a subclass of the `Literal` class). For example, the following codes create an evaluable literal says that x does not equal to y:

```
# the != operator is overridden for the MySymbol class and it returns an instance of the NEqua
# the NEqual is a subclass of ComparisonLiteral which inherits from the Literal class.
evaluable1 = (x != y)
# you can also create the NEqual explicitly.
evaluable1 = NEqual(x, y)
```

An evaluable literal has an `evaluate` method that evaluates the literal. If the boolean value of the literal can be determined solely on the literal itself, the method will return True or False, otherwise it returns a Z3 reference objects(which is tested for satisfiability with other evaluable literals in the partial subsumption algorithm).

MySymbol Class

MySymbol class is the base class for the literal arguments. There are 3 direct subclasses of the MySymbol class: Constant, Variable and FunctionSymbol.

Constant Class

A Constant instance stores a constant value which can only be a numeric type. For example, the following codes create a constant of value 1:

```
c = Constant(1)
```

When creating a Literal, you may pass a number directly to its constructor, for example, the following codes create a Literal named `p` and has three arguments 1, 2 and 3:

```
# When passing a constant to the Literal constructor, you can either create the Constant insta
p = Literal('p', [Constant(1), Constant(2), Constant(3)])
# or create the Constant instance implicitly by passing a number,as the constructor will conve
p = Literal('p', [1, 2, 3])
```

Variable Class

A Variable instance stores the name of a variable which can only be a string type. For example, the following codes create a variable named `var`:

```
v = Variable('var')
```

FunctionSymbol Class

Sometimes, we may want to pass some expressions as arguments to a literal. The most common ones are arithmetic operations such as plus, subtraction, etc. The FunctionSymbol Class is used to construct a function expression that can be passed as an argument to the Literal constructor. For your convenience, this tool has the basic arithmetic operators predefined. Here is a complete list of the predefined arithmetic operators: Plus(+), Subtract(-), Multiply(*) and Divide(/). You can write intuitive arithmetic expressions with these operators like the example below:

```
x = Variable('x')
y = Variable('y')
p = Literal('p', [x, y, x+y])
```

To create a custom function, you will need to define a function that takes MySymbol subclass instances as arguments, and register the function with the Function Class. For example, the following codes define a function named `modulo`:

```
# define the custom function
# the evaluate method returns the value the argument if it is a constant
# if the argument is a variable, it convert the variable to a Z3 Real type object
# if the argument is a function symbol, it calls the function of the symbol and returns the ou
def modulo(lhs, rhs):
    return lhs.evaluate() % rhs.evaluate()

# register the custom function with the Function class
Function.functionDict['modulo'] = modulo

x = Variable('x')
y = Variable('y')

# create a FunctionSymbol object using the custom function
# the FunctionSymbol constructor takes two positional arguments,
# the first is the name of the function,
# and the second is the list of its arguments
p = Literal('p', [x, y, FunctionSymbol('modulo', [x, y])])
```

Note: Users are not supposed to use the base class MySymbol, as the `partial_subsumption` method is dependent on the particular subclasses of MySymbol instances. Thus a variable should always be created with the Variable class, and a constant should always be created with the Constant class.

Examples

Please refer to the example.py.

References

1. Chakravarthy, Upen S., John Grant, and Jack Minker. "Logic-based approach to semantic query optimization." ACM Transactions on Database Systems (TODS) 15.2 (1990): 162-207.
2. Chang, Chin-Liang, and Richard Char-Tung Lee. Symbolic logic and mechanical theorem proving. Academic press, 2014.