Redundency Checking

November 27, 2018

1 Introduction

1.1 Purpose:

- **To verify that the addition of a new constraint F_k to a set of constraints $\{F_{1,\ldots,j}\}$ allows the inference of new formulas, which means F_k is not inferred by orginial set $\{F_1, \ldots, F_j\}$ otherwise it is the symptom that this new constraint is redundant.**

1.2 Definition:

- **Let $\{F_1, \ldots, F_j\}$ be a set of constraints. Let F_k be a new instruction. If $F_1 \wedge \ldots \wedge F_j \wedge F_k \not\models \perp$, *i.e*, $\{F_1, \ldots, F_j, F_k\}$ *is satisfiable*, then F_k is not redundant iff $F_1 \wedge \ldots \wedge F_j \wedge \neg F_k \not\models \perp$ *i.e*, $\{F_1, \ldots, F_j, \neg F_k\}$ *is satisfiable*.**
- If there exists $F_1 \wedge \ldots \wedge F_j \wedge \neg F_k \models \perp$, then, we have $F_1 \wedge \ldots \wedge F_j \models F_k$, which means F_k can be infered by $\{F_1, \ldots, F_j\}$, so F_k is redundant.

1.3 Conclusion:

- **Steps to check whether a new constraint F_k is redundant to a existing or not.**
- **1) Add F_k to existing set $\{F_{1,\ldots,j}\}$, check the satisfiability;**
- **2) Remove F_k and add $\neg F_k$, then, check the satisfiability;**
- **3) If both operations guarantee the satisfiability, the new constraint F_k is non-redundant.**

2 Implementation with Z3Py

```
In [5]: from z3 import *
        import re

In [6]: def redunCheckZ3(s, newCond):
            '''
            Description: pass through a set of constrains s,
            check the redundecny of new constraint newCond
            :type z3.z3.Solver: s
            :type z3.z3.BoolRef: newCond
```

```
        :rtype: string
        '''
        res = ''
        # Add the new constraint to existing set
        s.push()
        s.add(newCond)
        r1 = (s.check()==sat)
        # Remove the new constraint
        s.pop()
        # Add the negation of new constraint
        s.push()
        s.add(Not(newCond))
        r2 = (s.check()==sat)
        # Remove the constraint just added
        s.pop()
        # Check the redundency
        if r1 == True and r2 == True:
            res = 'Non-redundant'
        else:
            res = 'Redundant'
        return res
```

2.1 Example 1 - Arithmetic Operations

- **1) Variables: x as int, y as int**
- **2) Existing Statement:**
  - $i : (x + y)^2 > 10, x > 10, y > 10$
- **3) New Constraints:**
  - **New Constraint 1:** $x > 0$
  - **New Constraint 2:** $x < 100$

In [7]:
```
# Part 1: Defince constraints
# Define Variables
x = Int('x')
y = Int('y')
```

In [8]:
```
# Create an empty constraints set
s = Solver()
```

In [9]:
```
# Define the function t = (x+y)^2
t = simplify((x + y)**2, som=True)
```

In [10]:
```
# Add first constraint to body
s.push()
```

In [11]:
```
s.add(t>10)
```

In [12]:
```
# Display the body
s.assertions()
```

Out[12]:
```
[x*x + 2*x*y + y*y > 10]
```

In [13]:
```
# Add other constraints, x > 10, y > 10
s.push()
```

In [14]:
```
s.add(x>10,y>10)
```

In [15]:
```
# Display the body
s.assertions()
```

Out[15]:
```
[x*x + 2*x*y + y*y > 10, x > 10, y > 10]
```

2.1.1 Case 1. Redundant Constraint

- **New Constraint 1:** $x > 0$

In [16]:
```
# Part 2: Check new constraints
# Define first new constraint, x > 0
f = x > 0
```

In [17]:
```
# Check the redundancy
redunCheckZ3(s, f)
```

Out[17]:
```
'Redundant'
```

- **Explain: The existing constraint x > 10 always guarantee x > 0, so the new constraint x > 0 is redundant**

In [18]:
```
s.assertions()
```

Out[18]:
```
[x*x + 2*x*y + y*y > 10, x > 10, y > 10]
```

2.1.2 Case 2. Non Redundant Constraint

- **New Constraint 2:** $x < 100$

In [19]:
```
# Define first new constraint, x > 0
g = x < 100
```

In [20]:
```
# Check the redundancy
redunCheckZ3(s, g)
```

Out[20]:
```
'Non-redundant'
```

- **Explain: The new constraint x < 100 cannot be infered from the exisiting constraints, so it is Non-redundant**

2.2 Example 2 - Machine Arithmetic

- **1) Variables: x as bit vector(16 bits), y as bit vector(16 bits)**
- **2) Existing Statement:**
  - $i : x + y == 10, (x >= 3$ **or** $y >= 3)$
- **3) New Constraints:**
  - **New Constraint 1:** $x > 1$
  - **New Constraint 2:** $x > 3$

```
In [17]: x = BitVec('x', 16)
         y = BitVec('y', 16)
```

```
In [18]: s = Solver()
```

```
In [19]: t = (x + y == 10)
```

```
In [20]: s.push()
```

```
In [21]: s.add(t)
```

```
In [22]: s.assertions()
```

```
Out[22]: [x + y == 10]
```

```
In [23]: s.push()
```

```
In [24]: s.add(x>=3,y>=3)
```

```
In [25]: s.assertions()
```

```
Out[25]: [x + y == 10, x >= 3, y >= 3]
```

2.2.1 Case 1. Redundant Constraint

- **New Constraint 1:** $x > 1$

```
In [26]: f = x > 1
```

```
In [27]: redunCheckZ3(s, f)
```

```
Out[27]: 'Redundant'
```

2.2.2 Case 2. Non Redundant Constraint

- **New Constraint 2:** $x > 3$

```
In [28]: g = x > 3
```

```
In [29]: redunCheckZ3(s, g)
```

```
Out[29]: 'Non-redundant'
```

3 Implementation with YICES

```
In [35]: from yices import *
```

```
In [36]: def redunCheckYices(s, newCond):
             '''
             :param s: literals in body
             :param nfmal: the new constraint to check
             :return: res
             '''

             s.push()
             s.assert_formulas([newCond])
             status0 = ctx.check_context() == Status.SAT
             s.pop()
             s.push()
             s.assert_formulas([Terms.ynot(newCond)])
             status1 = ctx.check_context() == Status.SAT
             s.pop()
             if status0 and status1:
                 res = 'Non - redundant'
             else:
                 res = 'Redundant'
             return res
```

3.1 Example 1. Real Arithmetic

- **1) Variables: x as int, y as int**
- **2) Existing Statement:**
  - $i: 2x + y > 1, (x < 0 \textbf{ or } y < 0)$
- **3) New Constraints:**
  - **New Constraint 1:** $2x + y > 0$
  - **New Constraint 2:** $x < 1$

```
In [41]: cfg = Config()
         cfg.default_config_for_logic('QF_LRA')
         ctx = Context(cfg)   # type: Context

         real_t = Types.real_type()
         x = Terms.new_uninterpreted_term(real_t, 'x')
         y = Terms.new_uninterpreted_term(real_t, 'y')

         f0 = Terms.parse_term('(> (+ (* 2 x) y) 1)')  # type: object # x + y > 0
         f1 = Terms.parse_term('(or (< x 0) (< y 0))')  # x < 0 or y < 0
```

```
ctx.push()
ctx.assert_formulas([f0, f1])
```

3.1.1 Case 1. Redundant Constraint

- New Constraint 1: $2x + y > 0$

```
In [29]: newCond = Terms.parse_term('(> (+ (* 2 x) y) 0)')
         print(redunCheckYices(ctx, newCond))
```

```
Redundant
```

3.1.2 Case 2. Non Redundant Constraint

- **New Constraint 2:** $x < 1$

```
In [28]: newCond = Terms.parse_term('(< x 1)')
         print(redunCheckYices(ctx, newCond))
```

```
Non - redundant
```

3.2 Example 2. Bit-Vector

- **1) Variables: x as bit vector(32 bits) , y as bit vector(32 bits), z as bit vector(32 bits)**
- **2) Existing Statement:**
  - $i : x + y + z > 5, x > 1, y > 1, z > 1$
- **3) New Constraints:**
  - **New Constraint 1:** $x > 0$
  - **New Constraint 2:** $y > 3$

```
In [32]: cfg = Config()
         cfg.default_config_for_logic('QF_BV')
         ctx = Context(cfg)

         bv32_t = Types.bv_type(32)
         x = Terms.new_uninterpreted_term(bv32_t, 'x')
         y = Terms.new_uninterpreted_term(bv32_t, 'y')
         z = Terms.new_uninterpreted_term(bv32_t, 'z')

         constant = Terms.bvconst_integer(32, 5)
         f0 = Terms.bvgt_atom(Terms.bvadd(Terms.bvadd(x, y), z), constant)
         f1 = Terms.bvgt_atom(x, Terms.bvconst_integer(32, 1))
         f2 = Terms.bvgt_atom(y, Terms.bvconst_integer(32, 1))
         f3 = Terms.bvgt_atom(z, Terms.bvconst_integer(32, 1))
```

```
ctx.push()
ctx.assert_formulas([f0, f1, f2, f3])
```

```
Out[32]: True
```

3.2.1 Case 1. Redundant Constraint

- **New Constraint 1:** $x > 0$

```
In [33]: newCond = Terms.bvgt_atom(x, Terms.bvconst_integer(32, 0))

         print(redunCheckYices(ctx, newCond))
```

```
Redundant
```

3.2.2 Case 2. Non Redundant Constraint

- **New Constraint 2:** $y > 3$

```
In [34]: newCond = Terms.bvgt_atom(y, Terms.bvconst_integer(32, 3))
         print(redunCheckYices(ctx, newCond))
```

```
Non - redundant
```

4 Reference:

4.1 z3py-tutorial

4.2 z3 Guide

4.3 yices2_python_bindings