DCC888 - Static Analysis of Programs

Name: _____ **ID**: _____

Name: _____ **ID**: _____

By turning this exam in, I give my word that I have done it with my team only, on the understanding that we are allowed to consult any material publicly available, except material disclosed by other colleagues outside our team who are taking this course, or who have taken it in the past.

Goal: The goal of this exam is to implement a very simple optimization for a language of arithmetic and logical expressions. This optimization will avoid multiplying terms by zero. Our toy language will have the following instructions:

val(Num(n)) ↦ n	val(Add(s0, s1, E)) ↦ val(s0, E) + val(s1, E)	val(Eq(s0, s1, E)) ↦ val(s0, E) == val(s1, E)
val(Load(v, E)) ↦ E[v]	val(Mul(s0, s1, E)) ↦ val(s0, E) * val(s1, E)	val(Geq(s0, s1, E)) ↦ val(s0, E) ≥ val(s1, E)
val(Inv(s, E)) ↦ -1 * val(s, E)	val(Neg(s, E)) ↦ not(val(s, E))	val(Lth(s0, s1, E)) ↦ val(s0, E) < val(s1, E)

$$\frac{\text{val(cond, E)) } \mapsto \text{ True}}{\text{val(Cond(cond, e0, e1, E)) } \mapsto \text{ val(e0, E)}} \qquad \frac{\text{val(cond, E)) } \mapsto \text{ False}}{\text{val(Cond(cond, e0, e1, E)) } \mapsto \text{ val(e1, E)}}$$

Below, we have an implementation of this language in Python. You canfind this code at https://homepages.dcc.ufmg.br/~fernando/coisas/semiRing.txt:

```
class Num:
    def __init__(s, num):
        s.num = num
    def eval(s, env):
        return s.num

class Load:
    def __init__(s, name):
        s.name = name
    def eval(s, env):
        return env[s.name]

class UnOp:
    def __init__(s, src):
        s.src = src

class Inv(UnOp):
    def eval(s, env):
        return -1 * s.src.eval(env)

class Neg(UnOp):
    def eval(s, env):
        return not(s.src.eval(env))

class BinOp:
    def __init__(s, src0, src1):
        s.src0 = src0
        s.src1 = src1

class Add(BinOp):
    def eval(s, env):
        return s.src0.eval(env) + s.src1.eval(env)

class Mul(BinOp):
    def eval(s, env):
        return s.src0.eval(env) * s.src1.eval(env)

class Eq(BinOp):
    def eval(s, env):
        return s.src0.eval(env) == s.src1.eval(env)

class Geq(BinOp):
    def eval(s, env):
        return s.src0.eval(env) >= s.src1.eval(env)

class Lth(BinOp):
    def eval(s, env):
        return s.src0.eval(env) < s.src1.eval(env)

class Cond:
    def __init__(s, cond, e0, e1):
        s.cond = cond
        s.e0 = e0
        s.e1 = e1
    def eval(s, env):
        return s.e0.eval(env) if s.cond.eval(env) \
        else s.e1.eval(env)
```

If you want to build and evaluate some simple arithmetic expressions, you can use the tests below:

```
>>> Num(42).eval({})
42
>>> Load('a').eval({'a':1})
1
>>> Inv(Load('a')).eval({'a':1})
-1
>>> Neg(Load('a')).eval({'a':False})
True
>>> Add(Load('a'), Load('b')).eval({'a':1, 'b': 2})
3
>>> Mul(Load('a'), Load('b')).eval({'a':3, 'b': 2})
6
>>> Mul(Add(Load('a'), Load('b')), Load('b')).eval({'a':3, 'b': 2})
10
>>> Eq(Load('a'), Load('b')).eval({'a':1, 'b': 2})
False
>>> Eq(Add(Load('a'), Load('b')), Load('b')).eval({'a':0, 'b': 2})
True
>>> Geq(Load('a'), Load('b')).eval({'a':1, 'b': 2})
False
>>> Geq(Load('a'), Load('b')).eval({'a':2, 'b': 2})
True
>>> Lth(Load('a'), Load('b')).eval({'a':1, 'b': 2})
True
>>> Lth(Load('a'), Load('b')).eval({'a':2, 'b': 2})
False
>>> Cond(Load('c'), Load('a'), Load('b')).eval({'a':1, 'b':2, 'c':True})
1
```

Notice that if you know that the operand of a multiplication is zero, you don't need to evaluate the entire expression. In other words, you can add a conditional check to the expression, which either passes zero, or the product of a multiplication, depending on the value of some operands. See examples below:

```
# Original expression "a + b * c", without optimizations:
>>> env = {'a':2, 'b':0, 'c':3}
>>> Add(Load('a'), Mul(Load('b'), Load('c'))).eval(env)
2

# Optimized evaluation:
>>> add = Add(Load('a'), Mul(Load('b'), Load('c')))
>>> Cond(Eq(Load('b'), Num(0)), Load('a'), add).eval(env)
2
```

Another example of an optimized evaluation follows below. Notice that only `a` can null the entire expression if it is zero. The other variables, alone, cannot nullify the evaluation if they are zero:

```
# Expression "a * (b * c + d * a)"
# without optimizations:
>>> env = {'a':2, 'b':3, 'c':4, 'd':5}
>>> bc = Mul(Load('b'), Load('c'))
>>> da = Mul(Load('d'), Load('a'))
>>> add = Add(bc, da)
>>> Mul(Load('a'), add).eval(env)
44
```

```
# Optimized expressions:
>>> bc = Mul(Load('b'), Load('c'))
>>> da = Mul(Load('d'), Load('a'))
>>> add = Add(bc, da)
>>> cond = Cond(Eq(Load('a'), Num(0)), \
Num(0), Mul(Load('a'), add))
>>> cond.eval(env)
44
```

Question 1 [4 Point] Design an absorbing element analysis (AE), that finds “absorbing elements”. Absorbing values are values that nullify operations. An example is zero in multiplication. If E is an arithmetic expression, and AE(E) = {…, v, …}, then `v` MUST nullify E if `v` is zero. To design this analysis, you must provide transfer functions for every one of the ten types of instructions that we have in our simple language of arithmetic expressions.

Question 2 [2 Point] What is the semi lattice of your static analysis? Remember that a semilattice is built in terms of a set, a partial order, and a meet (or join) operator. Depending on the operation (meet or join) you might have either a top or a bottom element.

Question 3 [2 Point] What is the assymptotic complexity of your analysis, in terms of the number of instructions? In other words, what is the running time of AE(E), in terms of the size of E?

Question 4 [4 Point] Design a code transformation that adds conditional expressions to general expressions, to avoid multiplying terms if one of the operands is zero. You can try to design your optimization as a rewriting rule. For instance:

$$\frac{v \in AE(s0)}{opt(Mul(s0, s1)) \mapsto \ldots} \qquad opt(Num(n)) \mapsto Num(n)$$

Question 5 [4 Point] Implement the absorbing element analysis in Python, so that it works for arithmetic expressions written in our Python representation.

```
# Expression “a * (b * c + d * a)”
>>> bc = Mul(Load('b'), Load('c'))
>>> da = Mul(Load('d'), Load('a'))
>>> add = Add(bc, da)
>>> ae(Mul(Load('a'), add))
{‘a’}
```

```
# Expression “a * b * c”
>>> bc = Mul(Load('b'), Load('c'))
>>> abc = Mul(Load(‘a’), bc)
>>> ae(abc)
{‘a’, ‘b’, ‘c’}
```

If you like the object-oriented coding style, then feel free to add new methods to the classes that form arithmetic expressions. If you prefer a more functional implementation, you can either use pattern matching (available in Python 3.10) or you can use isinstance, e.g.:

```
>>> bc = Mul(Load('b'), Load('c'))
>>> isinstance(bc, Mul)
True
```

Question 6 [4 Point] Implement the opt transformation in Python. Like in Q5, you can either adopt a more object-oriented coding style, or a more function-oriented style. In the former case, you might have to add methods to the classes. On the latter, you might have to use some runtime type inspection.

To know more: this question has been inspired by the paper:

Guilherme V. Leobas, Fernando Magno Quintão Pereira: **Semiring optimizations: dynamic elision of expressions with identity and absorbing elements**. Proc. ACM Program. Lang. 4(OOPSLA): 131:1-131:28 (2020)