Project for the course MA220 :
Mathematical Logic and Computability

Demonstrating Chaitin's Incompleteness theorem in Python

Author:
Balaji R. Rao

December 6, 2012

Contents

1 Introduction

Chaitin's Incompleteness theorem [1] is statement about unprovability of statements about complexity of strings. The complexity of a string is defined to be the size of the smallest program that can produce it. Chaitin's theorem says that for a certain large enough natural number N, there exist strings of complexity greater than N which can't be proved to have complexity greater than N.

The proof is by contradiction and uses Berry's paradox.[2]. Chaitin himself has written LISP programs to demonstrate the proof [3].

In this project we attempt to demonstrate the proof in the Python language. Python is a good choice for such an undertaking, as it is a self-interpreted language, i.e, there is an implementation of a python in python itself [4].

In the first part of the project, we build structures for First-Order languages in Python. The second part of the project uses these structures to demonstrate the proof of Chaitin's incompleteness theorem.

2 First-Order Languages

2.1 Terms

We create the necessary structures for first order languages as classes in Python. Making use of inheritance we can represent the full structure of first order languages.

First, we begin by defining terms.

```
class Term (object) :
    def __init__(self):
        pass

class Variable (Term):
        def __init__(self, s=""):
            self._symbol = s

class Constant (Term) :
    def __init__(self, s):
        pass
```

Operations are represented as python functions that take an arbitrary number of arguments. Since our application of this to Chaitin's proof will include all possible strings as constants, we did not develop the Operation class beyond it's defenition.

```
class Operation (Term) :
    def __init__(self, function, degree, terms):
        self.f = function
        self.degree = degree
```

```
        self.terms = terms
        pass
```

2.2 Formulas

We define an abstract class for formulas from which all other classes which yield formulas are derived. We build a tree of formulas. This will be used during deduction using Modus-Ponens.

```
class Formula (object):
    def __post_order(self, f):
        for child in f.children :
            for gr_child in child.__iter__() :
                yield gr_child
        yield f

    def __iter__(self):
        return self.__post_order(self)

    def __init__(self, c = list()):
        self.children = list(c)
```

A 'Predicate' is a python function which returns boolean values. Hence it implicitly includes not only the definition of what the predicate does, but also the related axioms.

```
class Predicate :
    def __init__(self, function, degree):
        self.f = function
        self.degree = degree
        pass
```

We form subclasses to handle various ways formulas can recursively arise.

```
class AtomicFormula (Formula) :
    def __init__(self,predicate,  args, c=list()):
        self.predicate = predicate
        self.args = args
        Formula.__init__(self, c)

class Implies (Formula):
    def __init__(self, p, q):
        Formula.__init__(self)
        self.p = p
        self.q = q

class Iff (Formula):
```

```
    def __init__(self, p, q):
        Formula.__init__(self)
        self.p = p
        self.q = q

class Conjunction (Formula):
    def __init__(self, p, q):
        Formula.__init__(self)
        self.p = p
        self.q = q

class Disjunction (Formula):
    def __init__(self, p, q):
        Formula.__init__(self)
        self.p = p
        self.q = q

class Negation (Formula):
    def __init__(self, p):
        Formula.__init__(self)
        self.p = p

class Forall (Formula):
    def __init__(self, x, p):
        Formula.__init__(self)
        self.x = x
        self.p = p

class Exists (Formula):
    def __init__(self, x, p):
        Formula.__init__(self)
        self.x = x
        self.p = p
```

2.3 System

We represent a System as a python class which takes as arguments for its constructor, constants and axioms. The axioms are created using the above classes. we begin by including all the axioms as deduced statements and attaching it to a dummy root node.

```
class system :
    def __init__(self, constants, axioms):
        self.constants = constants
        self.deduced_statements = list(axioms)
        self.null = Formula(self.deduced_statements)
```

```
self.introduce_tautologies()
```

The function introduceTautologies will be explained later.

2.4 Deduction

Below is the main function, 'proofs' which does the work of deduction using Modus-Ponens. The statements deduced so far are maintained in a list, which also have an inherent tree structure.

While running through the list on one hand, we traverse the tree. On finding the required statement in the tree,we extend it and return the result. We use the python statement 'yield' instead of return, so that we can resume from where we left off.

We do this mainly to avoid generating the same pattern of formulas, which can happen if restart the search everytime we are asked for a proof. The 'yield' technique ensures that we go through the list and the tree completely every time.

```
def proofs (self) :
        while True :
            new_deductions = list()
            for d in self.deduced_statements :
                iter = self.null.__iter__()
                for i in iter :
                    if i.__class__ == Implies :
                        if i.p == d and
                        i.q not in self.deduced_statements:
                            d.children.append (i.q)
                            i.children.append (i.q)
                            yield i.q
                            new_deductions.append(i.q)
                           self.introduce_tautologies()
            self.deduced_statements.extend(new_deductions)
            if len(new_deductions) == 0 :
                break
```

The function introduce tautologies produces a minimum set of tautologies through which every tautology can be deduced See [5] This function is invoked every time a new deduction is introduced.

```
def introduce_tautologies(self):
      _new_deductions = list();

      for t in tautologies :
        iter = itertools.product
        (self.deduced_statements, t.__code__.co_argcount)
        for i in iter :
            _new_deductions.append(t (i))
```

```
self.deduced_statements.append(_new_deductions)
```

2.5 Logical Axioms

[5] provides a mimimun list of tautologies using which any tautology can be deduced using Modus Ponens. Since we have used classes to represent formulas, whose constructors use prefix notation, whereas logical formulas are usually given in infix notation, we used a parser to generate lambda functions from formulas given in infix notation. Soem of them are given here. For the whole list, see the code listing at the end.

```
tautologies =
[lambda p, q: Implies(p, p),
lambda p, q: Implies(p, Implies(q, p)),
lambda p, q: Implies(Negation(Negation(p)), p),
lambda p, q: Implies(p, Negation(Negation(p))),
lambda p, q: Implies(Negation(p), Implies(p, q)),
```

3 Chaitin's Incompleteness

We begin by defining an iterator for all possible strings.

```
def constant_generator() :
    i = 1
    allowed_chars = [chr(i) for i in range(255)]
    while True :
        iter =
        itertools.product(allowed_chars, repeat = i)
        for s in iter :
            yield s
        i = i + 1
```

The predicates are defined next, as python functions.

```
pool = Pool(processes=1)
def output_f(p, n, y) :
    #Predicate determining if a program halts in
    #n seconds and outputs y
    result = pool.apply_async(eval, (p,))
    if result.get(timeout = 1) == y :
        return True
    else
        return False

def is_nat_f(n) :
    #Predicate determining if a string is a
```

```
    #natural number
    return str.isdigit(n)

def length_greater_than_f(y, n) :
    #Predicate determining if length of a string is
    #less than n
    return (len(y) > n)

outputs = fol.Predicate(output_f, 3)
is_nat = fol.Predicate(is_nat_f, 1)
lgt = fol.Predicate(length_greater_than_f, 2)
```

Next, we create the system. Though we have not included any axioms here, any system which is complicated enough to express statements about complexity of strings will use the above predicates.

```
s = fol.system(constant_generator(), [])
```

Next, the actual code that generates the 'impossible' string.

```
for p in s.proofs() :
    if type(p) == Implies and
    type(p.q) == AtomicFormula and p.q.predicate == lgt :
        prog = p.q.args[0]
        n = p.q.args[1]

        p1 = p.p
        if type(p1) == Conjunction and
        type(p1.q) == AtomicFormula and
        p1.q.predicate == outputs and
        p1.q.args[0] == prog :
            y = p1.q.args[1]

            if type(p1.p) == AtomicFormula and
            p1.p.predicate == is_nat and
            p1.p.args[0] == n :
                print y
```

Here we look for the proof of the form $\text{nat}(n) \vee \text{outputs}(p, m, y) \implies \text{lgt}(p, n)$. This is done manually by examining the proof python-object. We examine it's structure through the class information. We use this to also extract the required output string and the 'proved' least-required length of a progream to produce it. This gives rise to a contradiction for a sufficiently large N, for example, one which is greater than the size of this program plus the python interpreter.

4 Code

4.1 fol.py

```
import itertools

class Term(object):
    def __init__(self):
        pass


class Variable(Term):
    def __init__(self, s=""):
        self._symbol = s


class Constant(Term):
    def __init__(self, s):
        pass


class Operation(Term):
    def __init__(self, function, degree, terms):
        self.f = function
        self.degree = degree
        self.terms = terms
        pass


class Formula(object):
    def __post_order(self, f):
        for child in f.children:
            for gr_child in child.__iter__():
                yield gr_child
        yield f

    def __iter__(self):
        return self.__post_order(self)

    def __init__(self, c=list()):
        self.children = list(c)


class Predicate:
    def __init__(self, function, degree):
        self.f = function
```

```
        self.degree = degree
        pass


class AtomicFormula(Formula):
    def __init__(self, predicate, args, c=list()):
        self.predicate = predicate
        self.args = args
        Formula.__init__(self, c)


class Implies(Formula):
    def __init__(self, p, q):
        Formula.__init__(self)
        self.p = p
        self.q = q


class Iff(Formula):
    def __init__(self, p, q):
        Formula.__init__(self)
        self.p = p
        self.q = q


class Conjunction(Formula):
    def __init__(self, p, q):
        Formula.__init__(self)
        self.p = p
        self.q = q


class Disjunction(Formula):
    def __init__(self, p, q):
        Formula.__init__(self)
        self.p = p
        self.q = q


class Negation(Formula):
    def __init__(self, p):
        Formula.__init__(self)
        self.p = p


class Forall(Formula):
```

```
    def __init__(self, x, p):
        Formula.__init__(self)
        self.x = x
        self.p = p


class Exists(Formula):
    def __init__(self, x, p):
        Formula.__init__(self)
        self.x = x
        self.p = p


tautologies = [lambda p, q: Implies(p, p),
               lambda p, q: Implies(p, Implies(q, p)),
lambda p, q: Implies(Negation(Negation(p)), p),
lambda p, q: Implies(p, Negation(Negation(p))),
lambda p, q: Implies(Negation(p), Implies(p, q)),
lambda p, q: Negation(Implies(p, Negation(q))),
lambda p, q: Implies(Conjunction(p, q),
        Negation(Implies(p, Negation(q)))),
lambda p, q: Implies(Negation(p), q),
lambda p, q: Implies(Disjunction(p, q),
        Implies(Negation(p), q)),
lambda p, q: Implies(p, Implies(Negation(q),
        Negation(Implies(p, q)))),
lambda p, q: Implies(Implies(p, q),
        Implies(Implies(Negation(p), q), q)),
lambda p, q: Implies(Implies(p, q),
        Implies(Negation(q), Negation(p))),
lambda p, q: Implies(Implies(p, q),
        Implies(Implies(q, p), Disjunction(p, q))),
lambda p, q: Implies(Disjunction(p, q),
        Implies(p, q)),
lambda p, q: Implies(Disjunction(p, q),
        Implies(q, p)),
lambda p, q: Implies(Implies(Negation(q), Negation(p)),
        Implies(Implies(Negation(q), p), q)),
lambda p, q, r: Implies(Implies(p, Implies(q, r)),
        Implies(Implies(p, q), Implies(p, r))),
]

class system:
    def __init__(self, constants, axioms):
        self.constants = constants
        self.deduced_statements = list(axioms)
```

```
        self.null = Formula(self.deduced_statements)


    def introduce_tautologies(self):
        _new_deductions = list();

      for t in tautologies:
        iter = itertools.product(
        self.deduced_statements, t.__code__.co_argcount)
        for i in iter:
            _new_deductions.append(t(i))

      self.deduced_statements.append(_new_deductions)

    def proofs(self):
        while True:
         new_deductions = list()
          for d in self.deduced_statements:
              iter = self.null.__iter__()
              for i in iter:
                  if i.__class__ == Implies:
                    if i.p == d and i.q not
                    in self.deduced_statements:
                        d.children.append(i.q)
                        i.children.append(i.q)
                        yield i.q
                        new_deductions.append(i.q)
                      self.introduce_tautologies()
          self.deduced_statements.extend(new_deductions)
          if len(new_deductions) == 0:
                break
```

4.2 chaitin.py

```
import itertools
import fol
from fol import Exists, Variable, Conjunction,
AtomicFormula, Implies, Constant
from time import sleep
from  multiprocessing import Pool

pool = Pool(processes=1)

def output_f(p, n, y):
    #Predicate determining if a program p outputs
    #y in less than or equal to n steps
```

```
    result = pool.apply_async(eval, (p,))
    if result.get(timeout=1) == y:
        return True
    else:
        return False


def is_nat_f(n):
    #Predicate determining if a
    #string is a natural number
    return str.isdigit(n)


def length_greater_than_f(y, n):
    #Predicate determining if length
    #of a string is less than n
    return (len(y) > n)


def constant_generator():
    i = 1
    allowed_chars = [chr(i) for i in range(255)]
    while True:
        iter = itertools.product(allowed_chars, repeat=i)
        for s in iter:
            yield s
        i = i + 1

outputs = fol.Predicate(output_f, 3)
is_nat = fol.Predicate(is_nat_f, 1)
lgt = fol.Predicate(length_greater_than_f, 2)

s = fol.system(constant_generator(), [])

for p in s.proofs():
    if type(p) == Implies and
    type(p.q) == AtomicFormula and p.q.predicate == lgt:
        prog = p.q.args[0]
        n = p.q.args[1]

        p1 = p.p
        if type(p1) == Conjunction and
        type(p1.q) == AtomicFormula
        and p1.q.predicate == outputs
        and p1.q.args[0] == prog:
            y = p1.q.args[1]
```

```
            if type(p1.p) == AtomicFormula and
            p1.p.predicate == is_nat
            and p1.p.args[0] == n:
                print y
```

4.3 logicalaxioms.py

```
# -*- coding: utf8 -*-
from pyparsing import Literal, Word, ZeroOrMore,
Forward, alphas, oneOf
pyparsGroup, operatorPrecedence, replaceWith

def Syntax():
    imp = Literal('=>').setParseAction
    (replaceWith("Implies"))
    conj = Literal('^').setParseAction
    (replaceWith("Conjunction"))
    disj = Literal('v').setParseAction
    (replaceWith("Disjunction"))
    iff = Literal(u'<=>').setParseAction
    (replaceWith("Disjunction"))

    op = imp | conj | disj | iff
    lpar = Literal('(').suppress()
    rpar = Literal(')').suppress()
    neg = Literal(u'!').setParseAction
    (replaceWith("Negation"))
    prop = Word(u"pqr")
    expr = Forward()
    atom = prop | Group(lpar + expr + rpar)
    expr << ((atom + ZeroOrMore(op + expr))
     | Group(neg + expr))
    return expr

def make_lambda(x):
    if len(x) == 1:
        if  x[0].__class__ == unicode:
            return x[0]
        else:
            return f(x[0])
    elif len(x) == 2:
        return "Negation (" + f(x[1]) + ")"
    elif len(x) == 3:
        return x[1] +
        " (" + f(x[0]) + "," + f(x[2]) + ")"
```

5 Limitations

We have attempted to include only those features of First Order Languages which are directly required for demonstrating the proof. The project does not impement functionality that handles free variables in formulas and the axiom of specialization, amonth other logical quantifier axioms, as give in [5], under the section "Syntactic Properties of Truth".

The axioms inlolving predicates used in Chaitin's proof are not worked out explicitly, but are taken for granted that they are expressed in the Python code corresponding to the function that implements the predicate. This limitation restricts ones understanding of how the limitation of finite axiom systems actually do come about, as one does not get an explicit description of the axioms involved.

References

[1] `http://en.wikipedia.org/wiki/Kolmogorov_complexity#Chaitin.27s_incompleteness_theorem`

[2] `http://en.wikipedia.org/wiki/Berry_paradox`

[3] `http://www.cs.auckland.ac.nz/~chaitin/lisp.html`

[4] `http://pypy.org`

[5] A Course in Mathematical Logic, Manin, I.U.I, 1977