Plumbum

Что такое Plumbum – это функциональный DSL (domain specific language) над частью языка LaTeX. Если быть точнее — над той частью, которая описывает как строятся все эти красивые формулы в документах. Plumbum позволяет преобразовать формулы, написанные на математическом языке в достаточно понятный кусок кода, который можно потом спокойно дебажить, применять различные познания в обратной разработке или же просто использовать как уже готовую написанную функцию.

Вопросы и ответы

1. Почему данный язык является DSL, а не языком общего предназначения?
    A. Потому что он никак не расширяет возможности далее того, что уже можно сделать в математических формулах и абстракциях. Например — реализация пользовательского ввода невозможна, ибо при создании формулы у нас нет такого понятия как пользователь в математике.
2. Почему данный язык является функциональным?
    A. Это следует из домена данного языка. Самое простое доказательство это неизменяемость переменных поскольку в математических формулах переменные не могут изменятся.
3. Почему именно LaTeX, а не свой язык с нуля или другой язык?
    A. Популярность. Данный язык является самым популярным для написания научных статей, так что готового инструментария для автоматической генерации формул, готовых визуальных и других редакторов и других программ уже существует в достаточно большом количестве. Плюсом он является open-source, так что не существует опасности закрытия стандарта исходного языка или других проблем с авторским правом (или причина отказа использования встроенного языка Microsoft Office).
4. Для чего может вообще потребоваться такого рода программа?
    A. Есть 2 крупных сценария использования (use-case). Первый это взаимодействие между теоретической частью институтами математики/физики/других точных наук и практической частью данного института (или вообще отдельного института вычислительных технологий). Если сейчас для такого взаимодействия нужен «переводчик», который одновременно хорошо понимает и теорию, и правила и методики построения компьютерных моделей, то при помощи plumbum можно уменьшить требуемые знания для такой роли (или вообще разнести данные функции по 2 сторонам взаимодействия и устранить посредника). Просто потому что довести хоть как-то корректно рабочий код до удобного вида гораздо проще и гораздо универсальнее задача (потому что она решаема частично без знания базовой модели), чем перевести модель в код. Кстати это и повлияло частично на выбор первых языков для трансляции — JS генерирует достаточно слабо перегруженный для человека код, а Python достаточно легко изучается. Вдобавок это перекликается со вторым сценарием — а что если нет возможности или поддержка взаимосвязи между звеньями слишком дорогая/долгая и программисту надо реализовать это с нуля, попутно разбираясь в модели. Тогда генерация такого кода может либо ускорить процесс путем возможности просто использования методик черного ящика, либо вообще подойти по параметрам и быть использованным уже для рабочих целей.

Примеры использования

Далее будут приведены примеры применения программы на различных формулах для языка Python и краткие пояснения к ним. Большая часть примеров является тестами, в таком случае будет указано имя файла.

Пример 1 — простое замыкание

Возьмем небольшой простой пример из файла ast.pb:

$$f(x) = (w(x,y) = x + y; g = w(x,*); g)\,; a = f(3); \models (a(4) \equiv 7)$$

В данном примере мы определяем некоторую функцию f как возвращающую некоторую функцию g. Данная функция g сама вызывает функцию w (которая сумма 2 аргументов) и фиксирует у неё первый параметр параметром функции f, а второй является параметром возвращаемой функции. Потом мы создаем переменную a и присваиваем ей значение. В конце написано следующее «в ранее описанной модели должно быть верно, что a(4) возвращает 7 в любом случае».

Посмотрим что за код был сгенерирован:

```
1  def _main():
2      def _f1():
3          def _f0(x):
4              def w(x,y):
5                  return (x+y)
6              g=(lambda _a2: w(x,_a2))
7              return g
8          def f(x):
9              return _f0(x)
10         a=f(3)
11         return (True if ((a(4)==7)) else exec('raise AssertionError(\'(a(4)==7)\')'))
12     _f1=_f1()
13     return _f1
```

Всё просто — т. к. режим работы был filer, то была сгенерирована функция main. Сначала надо описать функцию f. Сначала создаем замыкание отдельной функцией, потом именуем замыкание. В строках 4-5 описана функция w (поскольку она описана первой, то она и является первой), потом описываем g через лямбды. В строках 8-9 именуем ранее созданную безымянную функцию как f, в 10 строке объявляем переменную a и означиваем её, и в 11 задаем что при вызове функции (т. е. лениво) проверить истинность на модели.

Пример 2 – замыкания и множества

Для удобства чтения разобьем формулу из примера на 2 части. Начнем пример из файла closures.pb с определения функции f:

$$f(x) = (g(y) = 3 + (z = x + y; z)\,; h(y) = (z = x + y; z + 3)\,; w(y) = x + y + 3; \{g, h(3), (q = w(3); q)\})\,;$$

Технически тут описана функция, возвращающая множество из 3 элементов. Первый элемент множества это функция от одного переменного, остальные 2 это константы. Просто всё очень осложнено замыканиями.

Вторая часть примера определяет 2 переменные и просто проверяет модель на совпадение множеств:

$$r = f(4); t = r(0); \models (\{t(3), r(1), r(2)\} \equiv \{10, 10, 10\})$$

Полностью приводить сгенерированный код смысла нет (60 строк из которых большая часть определения функций), но на пару моментов стоит продемонстрировать. В продемонстрированной части кода показана функция f:

```
1  def _f5(x):
2      def _f0(y):
3          z=(x+y)
4          return z
5      def g(y):
6          return (3+_f0(y))
7      def _f1(y):
8          z=(x+y)
9          return (z+3)
10     def h(y):
11         return _f1(y)
12     def w(y):
13         return (x+y+3)
14     def _f4(x):
15         q=w(3)
16         return q
17     def _f2(x):
18         def _f3():
19             class _FarrN:
20                 def __init__(_s):_s.a=[g,h(3),_f4(x)]
21                 def __call__(s,i):return None if i<0 or i>=len(s.a) else s.a[i]
22                 def __repr__(s):return "<Pb %s>" % repr(s.a)
23                 def __eq__(s,o):return type(s).__name__==type(o).__name__ and s.a==o.a
24             return _FarrN()
25         _f3=_f3()
26         return _f3
27     return _f2(x)
28 def f(x):
29     return _f5(x)
```

Достаточно просто — замыкания продолжают порождать функции, порядок аргументов (например строка 6) сохраняется. Самое интересное происходит в строках 17-26. Тут происходит создание множества специального класса _FarrN, в котором определяется 4 встроенные функции: конструктор (__init__), вызов экземпляра класса (__call__), печать (__repr__), и сравнение на равенство (__eq__). Как видно в Python для хранения множества используется базовый тип list. Из особого стоит отметить что попытка выйти за границы множества не приводит к исключению, а нумерация начинается с 1 (для соответствия математическим моделям).

Пример 3 — словарь и рекурсия

Пример из файла dict.pb выглядит небольшим, но он является достаточно сложным для понимания:

$$\begin{cases} b_{\text{field1}} = 3 \\ b_{\text{field2}} = 5 \\ b_i = i \end{cases} ; * = b(\text{field3}, 7); * = b(\text{field4}, 9); \models (b_{\text{field3}} \equiv 7)$$

Что происходит? Фактически мы создаем структуру, которая является словарем. Т.е. Сначала мы создаем словарь с набором из 2 значений, потом добавляем ещё 2 значения и в части проверки модели проверяем что мы можем найти предпоследний элемент. Как искать элемент в словаре? Правильно — циклом... точнее рекурсией.

```
1  def _f1():
2      class _FrecN:
3          def __init__(_s):
4              _s.r={("field1",):3,("field2",):5}
5              class Pb(Exception):pass
6              _s.e=Pb
7          def __call__(_s,*_i):
8              while True:
9                  if len(_i)-1==1:
10                     _s.r[_i[:-1]]=_i[-1]
11                     return _i[-1]
12                 if _i in _s.r:return _s.r[_i]
13                 (i,)=_i
14                 _r=0
15                 try:_v=i
16                 except RecursionError:_r=1
17                 except _s.e:_r=1
18                 except Exception as e:
19                     if 'Pb' in e.__dict__:
20                         _i=e.vars
21                         continue
22                     else:_r=e
23                 if _r==1:raise _s.e('deep rec at ' + str(_i))
24                 if _r!=0:raise _r
25                 _s.r[_i]=_v
26                 return _v
27         def __repr__(s):return "<Pb rec %s>" % repr(s.r)
28     return _FrecN()
```

Класс _FrecN как раз отвечает за создание и поддержание рекурсии. В 4 строке в конструкторе мы задаем начальные значения (заодно видна реализация словаря — базовый тип dict). Теперь перейдем к функции __call__. Первый if отвечает за добавление новых элементов — если у нас число аргументов равно числу ключей + 1, то мы добавляем новый элемент. Потом в строке 12 мы просто делаем типичную для Python конструкцию по доставанию из dict элемента без исключения.

Возникает вопрос — а причем тут рекурсия? Фактически она тут не нужна, но из-за третьей строки (которая «если мы не нашли, вернуть ключ») и поскольку транслятор не знает, что там будет, но уже знает что не константа. Поэтому в наихудшем случае там будет рекурсия. Из плюсов — на простом примере можно увидеть механизмы защиты от бесконечных рекурсий (строка 16 и 23), других исключений, а также механизм мемоизации в строках 25-26.

Пример 4 – матрицы и map

Вторым важным элементом в математике являются матрицы. Поэтому в файле matrix.pb разбирается пример их создания:

$$m = \begin{pmatrix} 3 & 4 & 5 \\ 6 & 7 & 8 \end{pmatrix}; n = \begin{pmatrix} 1 \\ 2 \\ 3+1 \end{pmatrix}; o = \begin{pmatrix} 1 & 2 & 3+1 \end{pmatrix};$$

$$p = \{a + 1 \mid a \in m\}; r = \{m_{2,1}, n_1, o_2, p_{1,0}, p_{9,9}\}; \models (r \equiv \{8, 2, 4, 5, \otimes\})$$

Создаем матрицу m, вектор-столбец n и вектор-строку o, потом создаем матрицу через map-like метод, потом из выбранных элементов матриц.

Разберем код как создается первая матрица:

```
def _f1():
    class _Fmtx:
        def __init__(_s):
            _s.s=[3,2]
            _s.m=[3,4,5,6,7,8]
        def __call__(s,a,b=None):
            if b is None:
                if s.s[0]==1:
                    b=a
                    a=0
                elif s.s[1]==1:
                    b=0
                else:
                    raise Exception('No-vectored matrix expected two args')
            if a not in range(s.s[0]) or b not in range(s.s[1]):return None
            return s.m[a+b*s.s[0]]
        def __repr__(s):
            return '<Pb matrix\n%s\n>' % ('\n'.join(
                '\t'.join(str(e) for e in s.m[i*s.s[0]:(i+1)*s.s[0]])
                for i in range(s.s[1])
            ))
    return _Fmtx()
```

Как уже обычно — создаем класс _Fmtx. В конструкторе задаем размер матрицы и его элементы. В __call__ мы достаем элемент матрицы, обрабатывая случаи вектор-столбца, вектор-строки и выхода за пределы матрицы.

Теперь разберем конструкцию похожую на map:

```
def _f7():
    class _FmapN:
        def __init__(_s):
            _s.g=[m]
        def __call__(_s, *args):
            _v=_s.g[0](*args)
            if _v is None:return _v
            a=_v
            return (a+1)
        def __repr__(s):
            return '<Pb map %s>' % repr(s.g)
    return _FmapN()
```

Ничего сложного — храним изначальный элемент, при вызове проверяем на None (т. е. нет элемента), иначе модифицируем элемент перед выдачей. Поскольку в функциональных языках переменные статичные, то нет проблем с хранением по ссылке.