Tagged Union

Release 0.0.2

Jul 01, 2019

Contents:

CHAPTER 1

Examples

1.1 Successor Definition of Natural Numbers

```
from tagged_union import Unit, Self, tagged_union, match

@tagged_union
class Nat(object):
    O = Unit
    S = Self

    def __add__(self, other):
        return match(self, {
            Nat.O: lambda: other,
            Nat.S: lambda n: n + Nat.S(other)
        })

    def __sub__(self, other):
        return match(self, {
            Nat.O: lambda: self,
            Nat.S:
            lambda ns:
                match(other, {
                    Nat.O: lambda: self,
                    Nat.S: lambda no: ns - no,
                }),
        })

    def __mul__(self, other):
        return match(self, {
            Nat.O: lambda: Nat.O(),
            Nat.S: lambda ns: other + (other * ns), # x * y = y + (x - 1) * y
        })

    def __pow__(self, other):
```

(continues on next page)

(continued from previous page)

```
        return match(other, {
            Nat.O: lambda: Nat.S(Nat.O()), # forall x, x ** 0 = 1
            Nat.S: lambda no: self * (self ** no), # x ** y = x * x ** (y - 1)
        })

    def __lt__(self, other):
        return match(other, {
            Nat.O: lambda: False,
            Nat.S:
            lambda no:
                match(self, {
                    Nat.O: lambda: True,
                    Nat.S: lambda ns: ns < no,
                }),
        })

    def __ge__(self, other):
        return not self < other

    def to_int(self):
        return match(self, {
            Nat.O: lambda: 0,
            Nat.S: lambda n: 1 + n.to_int()
        })

    def __repr__(self):
        return repr(self.to_int()) + "%" + Nat.__name__

zero = Nat.O()

two = Nat.S(Nat.S(Nat.O()))
three = Nat.S(two)
four = Nat.S

a27 = three ** three
print(a27)

print(two * two * two * two * two * two * three)

print(two == Nat.S(Nat.S(Nat.O())))
print(two == a27)
```

1.2 Sorted, “Immutable” Binary Tree

```
from tagged_union import Unit, Self, tagged_union, match

@tagged_union
class BTree(object):
    branch = (Self, Self, object)
    leaf = Unit

    def add(self, data):
        return match(self, {
            BTree.leaf: lambda: BTree.branch(BTree.leaf(), BTree.leaf(), data),
```

(continues on next page)

(continued from previous page)

```
            BTree.branch: lambda l, r, d:
                BTree.branch(l.add(data), r, d) if data <= d \
                    else BTree.branch(l, r.add(data), d)
        })

    def __repr__(self):
        return match(self, {
            BTree.leaf: lambda: "",
            BTree.branch: lambda l, r, d:
                repr(l) + "/" + repr(d) + "\\" + repr(r),
        })

empty_btree = BTree.leaf()

print(empty_btree == BTree.leaf())

a = empty_btree.add(5)

print(a)

b = a.add(6)

print(b)

c = b.add(4)

print(c)
```

Python implementation of tagged unions.

Python tagged unions (aka sum types, algebraic data types, etc.) are a common tool in functional and functional style programming. This module provides a class decorator to concisely specify tagged unions (including recursively) as well as a match function for easily interacting with them.

To specify a class as a tagged union, decorate it with *tagged_union*. Note for Python2, this class must also inherit from *object*. This is no longer necessary in Python3. The possible members of the tagged union are specified as class attributes, equal to a type or tuple of types representing how that union member should be constructed. All tagged union members inherit from the orignal tagged union class, allowing common implementations inside this class.

Example

It is worth noting that python imports ignores identifiers which start with _, so if you wish to use the _ identifier as a wildcard for matching, it must be imported explicitly:

```
from tagged_union import _
from tagged_union import *
```

The following example creates a tagged union which has two members, *Foo* and *Bar*. *Foo* accepts no arguments as its constructor and *Bar* accepts an instance of MyTaggedUnion (either another *Bar* or a *Foo*):

```
@tagged_union
class MyU(object):
    Foo = Unit
    Bar = Self
```

(continues on next page)

(continued from previous page)

```
test = MyU.Bar(MyU.Bar(MyU.Bar(MyU.Foo())))
print(test)
```

It is then possible to use the match function against this new object. In this case, the *count* function counts how many MyU.Bar constructors appear in the object:

```
def count(test):
    return match(test, {
        MyU.Foo: lambda: 0,
        MyU.Bar: lambda x: 1 + count(x),
    })

print(count(test))
```

In its naive form, the match function can behave like a switch statement, if not used on a tagged union type. This also still supports _ for wildcarding matches:

```
def name_to_id(name):
    return match(name, {
        "Tom": lambda: 11,
        "Sarah": lambda: 12,
        _: lambda: -1,
    })

print(name_to_id("Tom"))
print(name_to_id("Michael"))
```

tagged_union.**Self**
: For defining recursive tagged unions. *Self* is replaced with the type of the tagged union class itself.

  Type type

tagged_union.**Unit**
: Use as a type for tagged union members who accept no arguments in their constructors. Also used internally as a sentinel to check that the correct number of arguments were given to a tagged union member's constructor.

  Type type

tagged_union._
: Used for wildcard matching. The corresponding dictionary value for the key of _ is called if the object being matched doesn't match any of the other dictionary keys.

  Type object

tagged_union.**match**(*union*, *branches*)
: Match statement for tagged unions (and other purposes).

  Allows matching of instances of tagged union members against their type. Can also just be used as a switch-like statement if used with other objects such as *int*. Uses the tagged union member's constructor arguments as the arguments to the matching branch's function.

  Parameters

  - **union** (*object*) – The object to be matches
  - **(dict of object** (*branches*) – function): A dict of which functions to call under the different matches. _ can be used for wildcard matching.

  Returns The result of calling the match's function.

  Return type object

tagged_union.**tagged_union**(*cls*)

Tagged Union class decorator.

Any members of the given class which are either types or tuples are converted into tagged union members, allowing members to be constructed from their identifiers.

Parameters **cls** (*type*) – The class to be converted to a tagged union

Returns The updated class with updated relevant members.

Return type type

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

t

Index

Symbols

M

S

T

U