Frameapp Documentation

Release 0.0

Maxim Avanov

Mar 18, 2018

Contents:

CHAPTER 1

Sum Types

1.1 Introduction

Sum type (or tagged union) is a data structure used to hold a value that could take on several different, but fixed, types. Only one of the types can be in use at any one time, and a tag explicitly indicates which one is in use. It can be thought of as a type that has several "cases", each of which should be handled correctly when that type is manipulated.[1]

Frameapp provides a limited subset of tagged unions, that have the same functionality as default Python Enums, plus additional features that allow each tag to hold a value, as well as a dispatching mechanism with consistency checks. You can think of Frameapp's Enums as a middle-ground between Python Enums and Rust Enums[2].

A SumType is defined with the following signature:

```
class MyType(SumType):
    VARIANT_TAG: TypeOfHoldedValue = associated_enum_value
```

The *VARIANT_TAG* and *associated_enum_value* behave exactly as Python Enums, yet the combination of *VARIANT_TAG* and *TypeOfHoldedValue* brings features of Rust Enums to this new data structure. Let's observe these properties through an example:

```
from typing import NamedTuple
from frameapp import SumType

class MyType(SumType):
    FOO: None = 'foo'
    BAR: None = 'bar'
    BAZ: int = 'baz'
```

Here we have defined a new tagged union *MyType* (which is another name for a sum type, remember), that consists of three distinct possibilities: *FOO*, *BAR*, and *BAZ*. In other words, instances of *MyType* is either *FOO* or *BAR* or *BAZ*, but not all of them at the same time.

[1] https://en.wikipedia.org/wiki/Tagged_union
[2] https://doc.rust-lang.org/book/second-edition/ch06-01-defining-an-enum.html

When the *associated_enum_value* (*'foo'*, *'bar'*, and *'buz'*) is not significant or is a string representation of a name of the tag, it can be omitted:

```
from typing import NamedTuple
from frameapp import SumType

class MyType(SumType):
    FOO: None
    BAR: None
    BAZ: int, int, None
```

The *TypeOfHoldedValue* (*None*, *None*, and *int* respectively) cannot be omitted, though. This definition basically says, that both *FOO* and *BAR* do not hold any value, yet the *BAZ* variant can hold an integer.

Now, let's observe the properties of the new type:

```
>>> MyType.FOO == MyType.BAR
False
>>> MyType.FOO == MyType.FOO()  # Tag instances that do not hold value are considered␣
↪the same as tags
True
>>> MyType.FOO() == MyType.BAR()  # ... yet different tags are still different tags
False
>>> MyType.BAZ(3) == MyType.BAZ
False
>>> MyType.BAZ(3) == MyType.BAZ(3)
True
>>> MyType.BAZ(3) == MyType.BAZ(4)
False
```

1.2 Contracts

Contracts define a fixed set of terms (classes, objects or functions), that every variant must have.

```
class LanguageWritingSystem(SumType):
    CYRILLIC: None
    LATIN: None
    HIEROGLYPH: None

    class Contract:
        AlphabetDatabase: models.Model
        CourseDatabase: models.Model
```

The SumType above represents a writing system that an application supports. The type says that at the moment there are three different types of writings, and the bounding contract states that each writing system has to have its own store for an alphabet and courses, no less and no more.

Frameapp makes sure that the application will be initialised only when there is an evidence that this contract is satisfied by all shards (at least, in type form). To provide such an evidence, we have to use a special decorator attached to every tag of the type:

```
@LanguageWritingSystem.CYRILLIC.bind(LanguageWritingSystem.Contract.AlphabetDatabase)
@LanguageWritingSystem.LATIN.bind(LanguageWritingSystem.Contract.AlphabetDatabase)
@LanguageWritingSystem.HIEROGLYPH.bind(LanguageWritingSystem.Contract.
↪AlphabetDatabase)
class Alphabet(models.Model):
```

```
    ...

@LanguageWritingSystem.CYRILLIC.bind(LanguageWritingSystem.Contract.CourseDatabase)
@LanguageWritingSystem.LATIN.bind(LanguageWritingSystem.Contract.CourseDatabase)
@LanguageWritingSystem.HIEROGLYPH.bind(LanguageWritingSystem.Contract.CourseDatabase)
class Course(models.Model):
    ...
```

We can bind a contract term of all three tags to the same object as in the example above, or to bind it to three separate objects, or to bind some of the tags to one object, and others to a separate set of objects - the decision depends on the particular situation. The only thing that matters is that all tags have complete versions (i.e. all terms are bound) of the same binding contract.

Apparently, for the case above, it's not optimal to have the same table for such different alphabets, so we might want to split it:

```
@LanguageWritingSystem.CYRILLIC.bind(LanguageWritingSystem.Contract.AlphabetDatabase)
class CyrillicAlphabet(models.Model):
    ...

@LanguageWritingSystem.LATIN.bind(LanguageWritingSystem.Contract.AlphabetDatabase)
class LatinAlphabet(models.Model):
    ...

@LanguageWritingSystem.HIEROGLYPH.bind(LanguageWritingSystem.Contract.
↪AlphabetDatabase)
class HieroglyphAlphabet(models.Model):
    ...

@LanguageWritingSystem.CYRILLIC.bind(LanguageWritingSystem.Contract.CourseDatabase)
@LanguageWritingSystem.LATIN.bind(LanguageWritingSystem.Contract.CourseDatabase)
@LanguageWritingSystem.HIEROGLYPH.bind(LanguageWritingSystem.Contract.CourseDatabase)
class Course(models.Model):
    ...
```

Regardless of our decision how data is persisted under the hood and how big a resulting difference between these alphabets, a hypothetical logic that relies on the *LanguageWritingSystem* remains the same.

```
def provide_alphabet_for(writing_system: LanguageWritingSystem) -> QuerySet:
    # here, `writing_system` is one of the tags of LanguageWritingSystem
    db = writing_system.Contract.AlphabetDatabase
    return db.objects.all()
```

It's worth highlighting that as soon as a tag is selected and passed to the function like in the example above, you no longer need to perform a conditional check of the *LanguageWritingSystem* - it is already matched, it is consistent, and all the contract terms available to you through the *Contract* attribute.

1.3 References

CHAPTER 2

Indices and tables

- genindex
- modindex
- search