Program with functors, applicatives and monads in Python

Chaur Wu

PyCon Latam 2023

About me

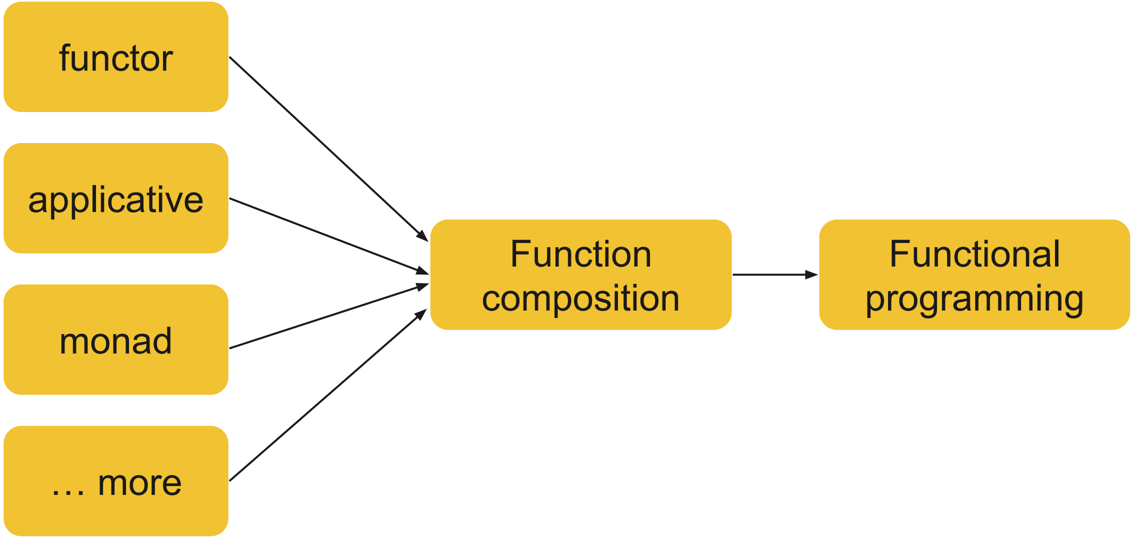
Chaur Wu (吳嘉二)

Software developer for over 20 years

Grew up in Taiwan

Based in the San Francisco bay area for the past 20 years

Why functors, applicatives and monads



composición de funciones

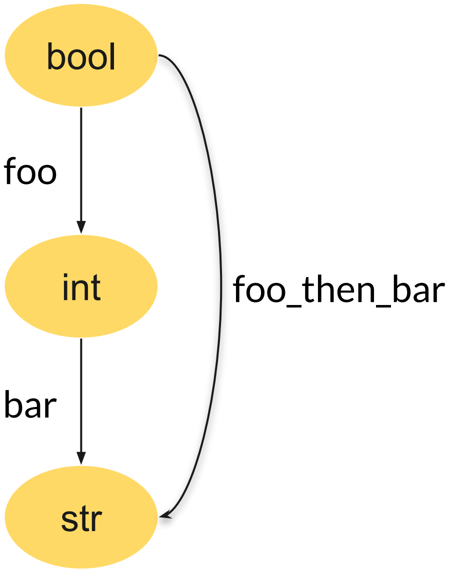
Rest of the talk

- Toy examples of functors, applicative, and monads with the funclift package (https://github.com/essentier/funclift)

- A more practical example that shows how to keep I/O side effects out of the functional core. The example will use free monad to build and interpret a simple DSL (domain specific language) for accessing files.

Composing pure functions

```python
def foo(b: bool) -> int:
    return (2 if b else 3)


def bar(n: int) -> str:
    return 'h' * n


def foo_then_bar(b: bool) -> str:
    return bar(foo(b))
```



Partial functions

Partial functions

```
def ten_mod_by(n: int) -> int:
    return 10 % n
```

What’s the problem?

Partial functions

```python
def ten_mod_by(n: int) -> int | None:
    if n == 0:
        return None

    return 10 % n
```

What’s the problem?

Partial functions (not very composable)

```python
def remainder_in_text(r: int) -> str:
    return 'remainder is ' + str(r)

def ten_mod_by_in_text(x: int) -> str | None:
    r = ten_mod_by(x)
    if r:
        return remainder_in_text(r)
    else:
        return None
```

Partial functions (Option)

```
from funclift.types.option import Option, Nothing, Some


def ten_mod_by(n: int) -> Option[int]:
    if n == 0:
        return Nothing()

    return Some(10 % n)
```

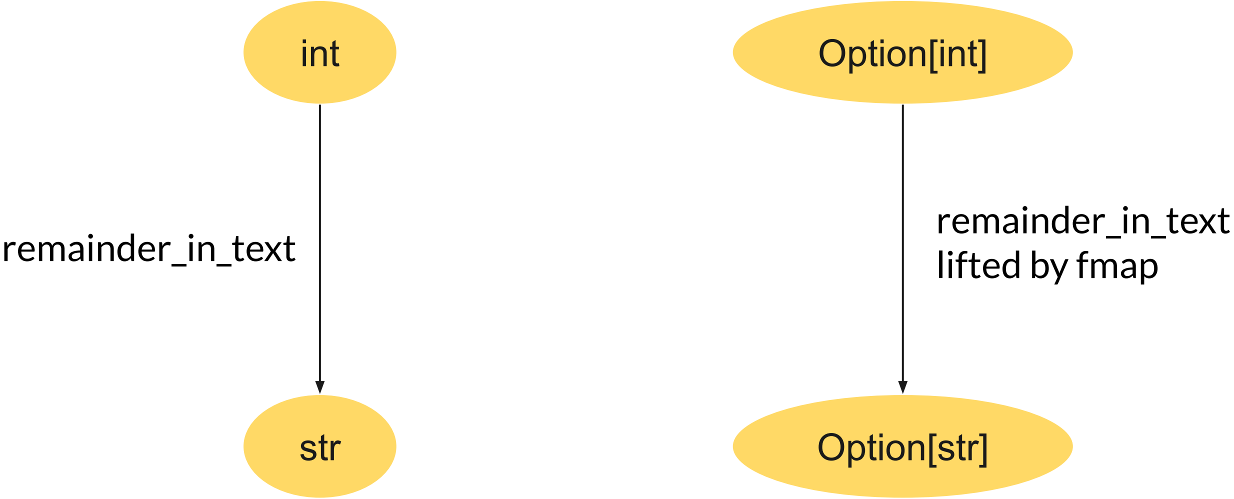
Partial functions (functor)

```
def remainder_in_text(r: int) -> str:
    return 'remainder is ' + str(r)


def ten_mod_by_in_text(x: int) -> Option[str]:
    r = ten_mod_by(x)
    return r.fmap(remainder_in_text)
```

Functor class

```
class Functor(Generic[F, A], Protocol):

    def fmap(self, f: Callable[[A], B]) -> Functor[F, B]:
        ...
```

A functor is a mapping from one category to another



functor laws

- The implementation of a functor needs to satisfy some laws, which can not be type checked.

- Use a property-based testing library such as hypothesis (https://github.com/HypothesisWorks/hypothesis) to test a functor implementation

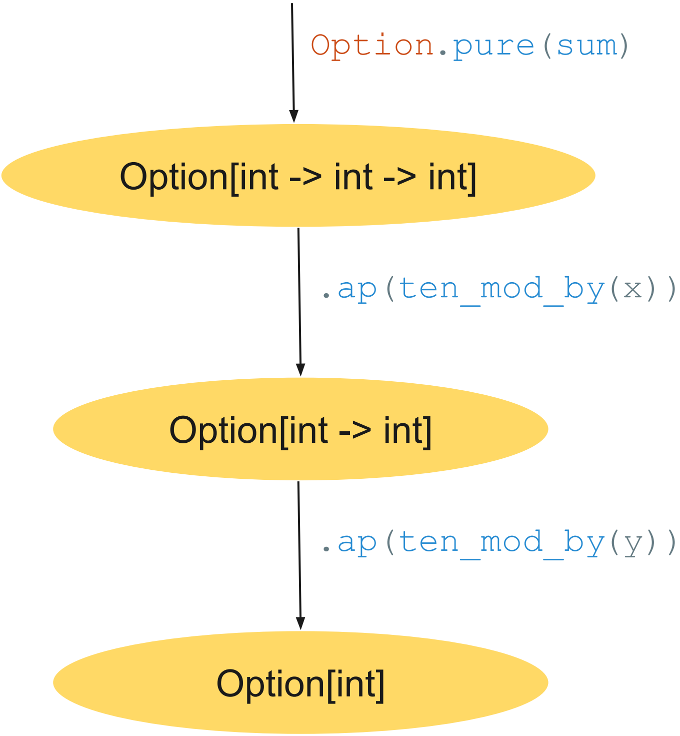
- An example (https://github.com/essentier/funclift/blob/main/tests/funclift/option_test.py)

Partial functions (not very composable)

```python
def sum(a: int, b: int) -> int:
    return a + b

def sum_mod_bys(x: int, y: int) -> int | None:
    rx = ten_mod_by(x)
    ry = ten_mod_by(y)
    if rx and ry:
        return sum(rx, ry)
    else:
        return None
```

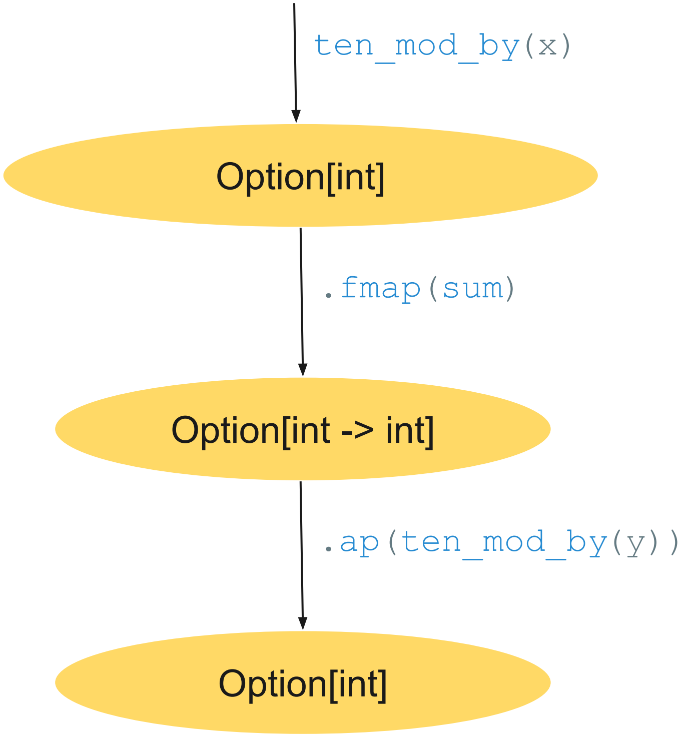
Partial functions (applicative)

```
@curry
def sum(a: int, b: int) -> int:
    return a + b

def sum_mod_bys(x: int, y: int) -> Option[int]:
    return Option.pure(sum) \
        .ap(ten_mod_by(x)) \
        .ap(ten_mod_by(y))
```


Option.pure(sum)
Option[int -> int -> int]
.ap(ten_mod_by(x))
Option[int -> int]
.ap(ten_mod_by(y))
Option[int]

Partial functions (alternative usage of an applicative)

```
def sum_mod_bys(x: int, y: int) -> Option[int]:
    return ten_mod_by(x) \
        .fmap(sum) \
        .ap(ten_mod_by(y))
```



ten_mod_by(x)
Option[int]
.fmap(sum)
Option[int -> int]
.ap(ten_mod_by(y))
Option[int]

Applicative class

```
class Applicative(Functor[F, A], Protocol):

    @staticmethod
    def pure(a: A) -> Applicative[F, A]:
        ...

    def ap(self: Applicative[F, Callable[[C], E]],
           a: Applicative[F, C]) -> Applicative[F, E]:
        ...
```

Partial functions (more modulo arithmetic)

```
def seven_mod_by(n: int) -> int | None:
    if n == 0:
        return None
    return 7 % n

def three_mod_by(n: int) -> int | None:
    if n == 0:
        return None
    return 3 % n
```

Partial functions (not very composable)

```
def mod_bys(n: int) -> int | None:
    r10 = ten_mod_by(n)
    if r10:
        r7 = seven_mod_by(r10)
        if r7:
            return three_mod_by(r7)
    return None
```

Partial functions (modulo arithmetic with Option)

```
def seven_mod_by(n: int) -> Option[int]:
    if n == 0:
        return Nothing()
    return Some(7 % n)

def three_mod_by(n: int) -> Option[int]:
    if n == 0:
        return Nothing()
    return Some(3 % n)
```

Partial functions (monad)

```
def mod_bys(n: int) -> Option[int]:
    return ten_mod_by(n) \
        .flatmap(seven_mod_by) \
        .flatmap(three_mod_by)
```

Partial functions (monad do-notation)

```
from funclift.fp.monad_runner import run_monads


def mod_bys(n: int):
    r10 = yield ten_mod_by(n)
    r7 = yield seven_mod_by(r10)
    return three_mod_by(r7)


monads = mod_bys(10)
result = run_monads(monads)
```

Monad class

```
class Monad(Applicative[F, A], Protocol):

    @abstractmethod
    def flatmap(self, f: Callable[[A], Monad[F, B]]) -> Monad[F, B]:
        ...
```

Summary of effectful function compositions

f :: A -> B g :: B -> C	functor	f′ :: F[A] -> F[B] g′ :: F[B] -> F[C]
f :: A -> (B -> C)	applicative	f′ :: F[A] -> (F[B] -> F[C])
f :: A -> F[B] g :: B -> F[C]	monad	g.f :: A -> F[C]

More ways for composing effectful functions

- Contravariant functors (Predicate)
- Bifunctor (Either)
- Profunctor (Star, Costar)

IO class for input/output side effects

Functions that perform IO actions

```
def my_print_v1(message: str) -> None:
    print(message)
```

```
def my_print_v2(message: str) -> IO[None]:
    return IO(lambda: print(message))
```

IO actions

```
@curry
def sum(a: int, b: int) -> bool:
    return a + b

def is_even(n: int) -> bool:
    return n % 2 == 0

def get_number() -> IO[int]:
    return IO(lambda: int(input('enter a number: ')))

def print_result(num_even: bool) -> IO[None]:
    return IO(lambda: print('is even: ', num_even))
```

Compose IO actions

```
num1 = get_number()

num2 = get_number()

num = IO.pure(sum).ap(num1).ap(num2)

num_even = num.fmap(is_even)

program = num_even.flatmap(print_result)

program.unsafe_run()
```

Compose IO actions (point-free style)

```
get_number() \
     .fmap(sum) \
     .ap(get_number()) \
     .fmap(is_even) \
     .flatmap(print_result) \
     .unsafe_run()
```

Compose IO actions (do-notation)

```
def create_program_monads ():
    num1 = yield get_number()
    num2 = yield get_number()
    num = sum(num1, num2)
    num_even = is_even(num)
    _ = yield print_result(num_even)
    return IO.pure(None)

monads = create_program_monads ()
program = run_monads(monads)
program.unsafe_run()
```

Writer class for mutating external states

Functions that mutate external states

```
def my_writer_v1() -> int:
    log.debug('hello')
    return 42

def my_writer_v2() -> Writer[str, int]:
    return Writer(42, 'hello')
```

Writer actions

```python
def get_number_with_log(n: int) -> Writer[list[str], int]:
    return LogWriter.pure2(n, [f'got number {n}'])

@curry
def sum(a: int, b: int) -> bool:
    return a + b

def is_even(n: int) -> bool:
    return n % 2 == 0

def log_is_even(b: bool) -> Writer[list[str], bool]:
    return LogWriter.pure2(b, [f'sum is even: {b}'])
```

Compose Writer actions

```
num1 = get_number_with_log(5)
num2 = get_number_with_log(3)
num = LogWriter.pure(sum).ap(num1).ap(num2)
num_even = num.fmap(is_even)
num_even_logged = num_even.flatmap(log_is_even)
print(num_even_logged)
```

```
Writer(value=True, written=['got number 5', 'got number 3', 'sum is even: True'])
```

Effects and types

Partial functions	Option, Either, Validated
Input / output	IO
write external states	Writer
Read external states	Reader
Read and write external states	State
Nondeterministic values	List

Monad transformer

```
def get_number() -> IO[int]:
    return IO(lambda: int(input('enter a number: ')))
```

What if the user does not enter a number? We need IO[Option[int]]

Monad transformer (OptionT)

```python
from funclift.types.optiont import OptionT

def get_number_io() -> IO[str]:
    return IO(lambda: input('enter a number: '))

def create_program_monads() -> OptionT[IO, int]:
    num_str = yield OptionT.lift(get_number_io())
    try:
        return Some(int(num_str))
    except ValueError:
        return Nothing()
```

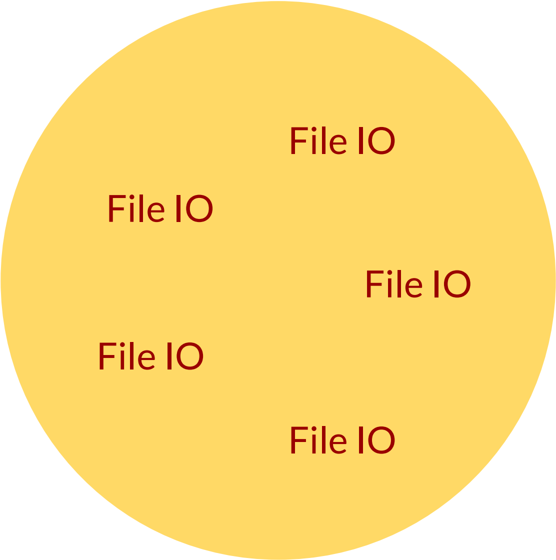
Monad transformer

```
monads = create_program_monads()
program = run_monads(monads)
result = program.run().unsafe_run()
print(result)
```

A more practical example

```python
class TextFile:
    @staticmethod
    def read(filename: str) -> str:
        with open(filename, "r") as file:
            return file.read()

    @staticmethod
    def write(filename: str, text: str) -> None:
        with open(filename, "w") as file:
            file.write(text)


filename = 'hello.txt'
TextFile.write(filename, 'Hello PyCon Latam')
content = TextFile.read(filename)
```

What are the problems?

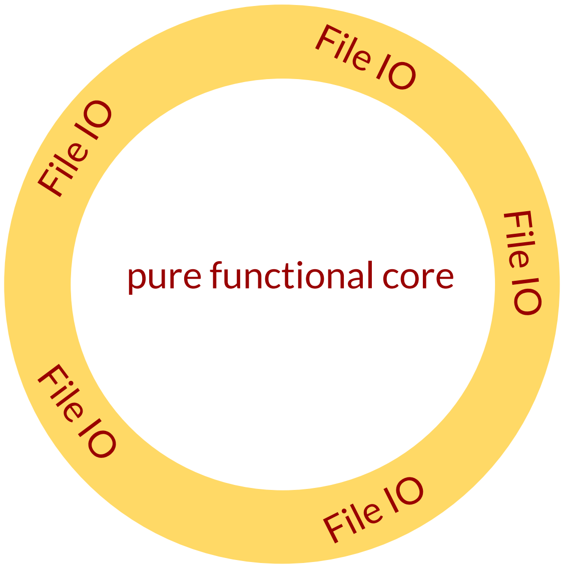


Side effects all over the place

Breaks referential transparency

> “Effects are good; side effects are bugs.”
>
> – Rob Norris

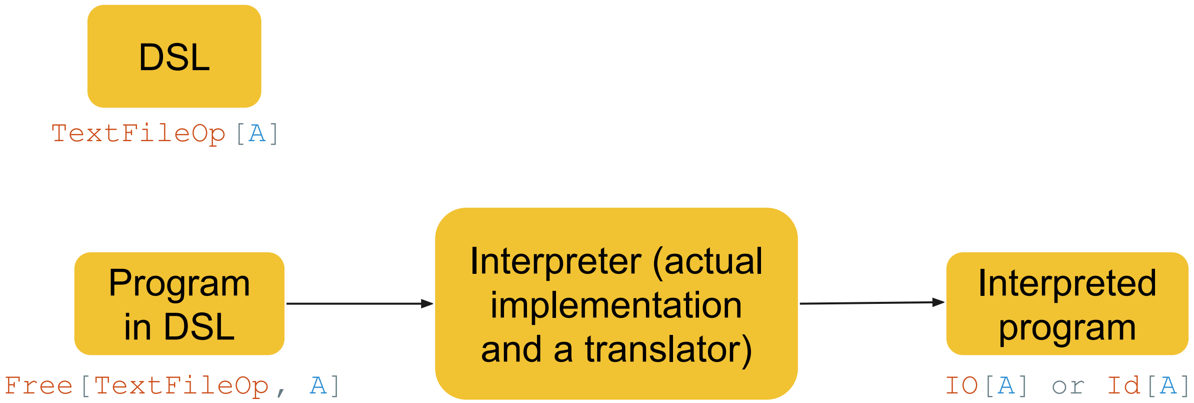
Solution



Use Free Monad to build a simple DSL for reading and writing files.

Interpret the DSL as IO effects.

All of that without side effects in Python and with type hints by the end of the talk



DSL

`TextFileOp[A]`

Program in DSL

`Free[TextFileOp, A]`

Interpreter (actual implementation and a translator)

Interpreted program

`IO[A] or Id[A]`

DSL

```
class TextFileOp(ABC, Generic[A]):
    def lift(self) -> Free[TextFileOp, A]:
        return Free.liftm(self)

@dataclass
class Read(TextFileOp[str]):
    filename: str

@dataclass
class Write(TextFileOp[None]):
    filename: str
    text: str
```

Program in DSL

```
def create_app_dsl():
    filename = 'hello.txt'
    _ = yield Write(filename, 'Hello PyCon Latam 2023').lift()
    contents = yield Read(filename).lift()
    return contents


app_dsl = create_app_dsl()
program = run_monads(app_dsl)
io_effects = program.foldmap(TextFileOpToIO())
content = io_effects.unsafe_run()
```

IO Interperter (actual implementation)

```python
class TextFileIO:
    @staticmethod
    @io_effect
    def read_effect(op: Read) -> str:
        with open(op.filename, "r") as file:
            return file.read()

    @staticmethod
    @io_effect
    def write_effect(op: Write) -> None:
        with open(op.filename, "w") as file:
            file.write(op.text)
```

Natural transformation mapping TextFileOp[A] to IO[A]

```
class TextFileOpToIO():
    def mempty(self, a: A) -> IO[A]:
        return IO.pure(a)

    def apply(self, op: TextFileOp[A]) -> IO[A]:
        match op:
            case Read():
                return TextFileIO.read_effect(op)
            case Write():
                return TextFileIO.write_effect(op)
```

Mock/Stub Interperter (actual implementation)

```
class TextFileMock():
    contentsMap: dict[str, str] = {}

    @staticmethod
    @id_effect
    def read_effect(op: Read) -> str:
        return TextFileMock.contentsMap[op.filename]

    @staticmethod
    @id_effect
    def write_effect(op: Write) -> None:
        TextFileMock.contentsMap[op.filename] = op.text
```

Natural transformation mapping TextFileOp[A] to Id[A]

```
class TextFileOpToMock():
    def mempty(self, a: A) -> Id[A]:
        return Id.pure(a)

def apply(self, op: TextFileOp[A]) -> Id[A]:
    match op:
        case Read():
            return TextFileMock.read_effect(op)
        case Write():
            return TextFileMock.write_effect(op)
```

muchas gracias