Multiple & Predicative Dispatch

Multiple and Predicative Dispatch

In any programming language, dispatch is a fundamental language constructs.

A first example shows the `gnosis-dispatch` library; we'll work backwards towards theoretical ideas underlying it.

david.mertz@seiu.org

pypi.org/project/gnosis-dispatch/

Multiple and Predicative Dispatch

SEIU is a labor union representing 2 million workers in the United States, Puerto Rico, and Canada, founded in 1921.

We fight for a just society where all workers are valued and all people respected—no matter where we come from or what color we are; where all families and communities can thrive; and where we leave a better and more equitable world for generations to come.

A note on the illustrations

As Principal Engineer of SEIU, I am architect of a computer system, named after 19th century computer pioneer Ada Lovelace.

Ada ingests data from union locals; I named its automated background processes as *Jacquard looms*.

Watercolor portrait of Ada King, Countess of Lovelace, c. 1840, possibly by Alfred Edward Chalon

Public domain

https://en.wikipedia.org/wiki/File:Ada_Lovelace_portrait.jpg

Multiple and Predicative Dispatch

We would like to analyze information about nations, using functions like `income()`, `morbidity()`, `exports()`, or `demographic()`.

There are many nations, and many ways to obtain information depending on which nation it is (different national databases, different APIs to query information, etc.).

Object-oriented programming offers the *delegation pattern*. I think there is a better way.

Multiple and Predicative Dispatch

In the scenario I describe, some developers have implemented some of these functions, for some nations. However, there are many gaps in what the initial developers can provide.

They decide to publish a Python module called `world_data` that exposes the data and algorithms they know, but also allows for easy extension by users and by other developers.

Multiple and Predicative Dispatch

Hypothetical `nations` namespace, extended:

```
from world_data import Place, nations

@nations
def demographic(place: Place & place.iso_3166 == "ZA"):
    url = "https://statssa.gov.za/"
    api_key = get_api_key(credential)
    …
    return demographic_data

@nations
def demographic(place: str & place == "Lesotho"):
    host = "database.gov.ls"
    user = "research_centre"
    with psycopg.connect(host=host, user=user) as conn:
        …
    return demographic_data
```

Multiple and Predicative Dispatch

Each function in `nations` is an implementation that will be selected where it best matches.

```
@nations
def demographic(place: str & place == "Lesotho"
                source: Authority):
    # Choose among references sources
    nation_info = source.fetch_raw_data(place)
    # …
    return demographic_data
```

The Lesotho government provides information; also do United Nations, African Union, and World Bank.

Matching *more* annotations “wins” over fewer. Types and predicates choose an implementation.

Multiple and Predicative Dispatch

After binding implementations, we'll have e.g.:

```
>>> str(nations)
'nations with 16 functions bound to 1348 implementations'
>>> nations.describe()
nations bound implementations:
(0) income
    place: Any ∩ place.name == "Algeria"
(1) income
    place: Place ∩ place.iso_3166 == "ZA"
 …
(0) demographic
    place: str ∩ place == "Lesotho"
(1) demographic
    place: str ∩ place == "Lesotho"
    source: Authority
 …
```

Multiple and Predicative Dispatch

Type systems & flow control are purportedly orthogonal. Consider function overloading.

```
#include <iostream>

int vol(int s) {                  // Volume of cube
  return s * s * s; }
double vol(double r, int h) { // Cylinder
    return 3.14159 * r * r * static_cast<double>(h); }
long vol(long l, int w) {      // Square prism
    return l * w * w; }

int main() {
    std::cout << "Cube vol(5) = "     << vol(5)
      << "\nCylinder vol(2.5, 7) = " << vol(2.5, 7)
      << "\nPrism vol(3l, 6) = "     << vol(3l, 6)
      << std::endl; }
```

Multiple and Predicative Dispatch

Dispatching to the name `vol()` is flow control based on the types of arguments.

```
% g++ -o function-overload function-overload.cpp
% ./function-overload
Cube     vol(5) = 125
Cylinder vol(2.5, 7) = 137.441
Prism    vol(3l, 6) = 108
```

Many programming languages support function overloading. Most commonly based on *type* and *number* of arguments.

Often we think of data types as `uint8`, `int32`, `float64`, `complex128`, `string`, or even `[6]int16`.

There's nothing fundamental about, e.g., `uint8` being numbers from 0 to 255. A number between 1 and 100 is just as coherent.

We inherit historical accidents of how computer hardware has developed (8-bit bytes, 32-bit words, binary encoding, etc.).

Multiple and Predicative Dispatch

In a last non-Python example, let's look at Haskell code for a bounded integer type.

```
newtype SmallInt = SmallInt Integer deriving Show
smallInt :: Integer → SmallInt
smallInt n
  | 0 < n && n < 100 = SmallInt n
  | n > 99 = SmallInt 100 -- clip to 100
  | otherwise = error "Not positive"

main = do
putStrLn "Numbers between 1 and 100"
print (smallInt 25)
print (smallInt 120)
print (smallInt (-6))
```

```
% runhaskell smallInt.hs
Numbers between 1 and 100
SmallInt 25
SmallInt 100
SmallInt smallInt.hs:
   Not positive
```

Multiple and Predicative Dispatch

The equivalent code in Python:

```python
class SmallInt(int):
    def __new__(cls, value):
        if not isinstance(value, int):
            raise ValueError("SmallInt must be an integer")
        elif value < 1:
            raise ValueError("SmallInt must be positive")
        elif value > 100:
            value = 100  # Clip to maximum value of 100
        return super().__new__(cls, value)

try:
    print(SmallInt(25))  # ⟶ 25
    print(SmallInt(120)) # ⟶ 100
    print(SmallInt(-6))  # Error
except Exception as err:
    print(err)
```

```
% python small-int.py
25
100
SmallInt must be positive
```

Multiple and Predicative Dispatch

Initializing a value is of limited use. Other operations might also remain in the domain.

```
class SmallIntAdd(SmallInt):
    def __add__(self, other):
        return SmallIntAdd(super().__add__(other))

print(SmallIntAdd(25) + SmallInt(30)) # ⟶ 55
print(SmallIntAdd(25) + 150) # ⟶ 100
print(150 + SmallIntAdd(25)) # ⟶ 175
```

```
% py small-int-add.py
55
100
175
```

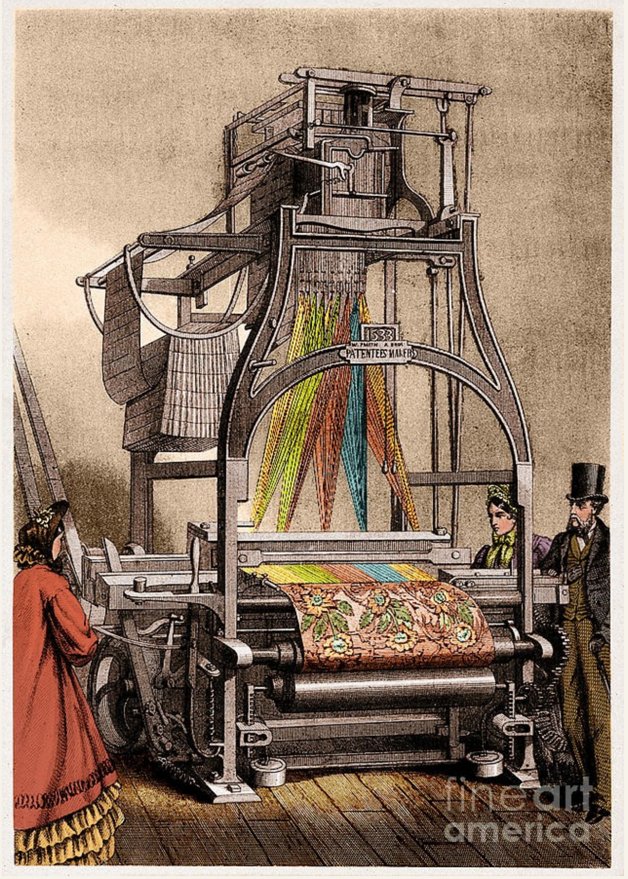
The example only addresses addition of a right-side argument. We might also implement `__radd__`, `__mul__`, etc.

Multiple and Predicative Dispatch

An abstract perspective re-frames the meaning of “data type” as “the collection of all values satisfying certain predicates.”

An `int16` is any integral value between -32,768 and 32,767.

A predicate being easy to represent with existing computer hardware isn't inherent to the meaning a data type.



My library, `gnosis-dispatch`, provides one way of structuring code. Let's consider other ways of organizing flow control before we return to it.

Let's look at several functions that can test for the primality of numbers, each suitable for different contexts.

Multiple and Predicative Dispatch

Each of these functions *does* the same thing, but within different domains.

```
def is_medium_prime(n: int):
  "Check prime factors n < √2³²"
  for p in primes_16bit:
    if p > sqrt(n): return True
    if n % p == 0: return False
  return True
```

```
def is_small_prime(n: int):
  "Check for primes n < 2¹⁶"
  return n in primes_16bit
```

```
def miller_rabin_prime(n):
  "Miller-Rabin probabilistic"
  # … implementation …
```

```
def gaussian_prime(c: complex):
  "Check for Gaussian prime"
  # … implementation …
```

```
def agrawal_kayal_saxena(n):
  "AKS deterministic test"
  # … implementation …
```

Multiple and Predicative Dispatch

How can `is_prime()` cover all our cases?

One way is to use `if/elif/else`.

```
def is_prime(num):
    if instance(num, complex):
        return gaussian_prime(num)
    elif isinstance(num, int):
        if num ⩽ 0: raise ValueError
        elif num < 2**16:
            return is_small_prime(num)
        elif num < 2**32:
            return is_medium_prime(num)
        else:
            return miller_rabin_prime(num)
    else:
        raise ValueError
```

Multiple and Predicative Dispatch

The `if/elif` code intermixed tests of data types with tests of predicates. We *can* express it purely in terms of types.

```
def is_prime(num):
    match num:
        case SmallInt(m):
            return is_small_prime(m)
        case MediumInt(m):
            return is_medium_prime(m)
        case int(m):
            return miller_rabin_prime(m)
        case complex() as c:
            return gaussian_prime(c)
        case _:
            raise ValueError
```

Multiple and Predicative Dispatch

The types-only match is perhaps cheating; we've embedded predicates in the type definitions. Should we just use methods?

```python
class SmallInt(int):
    def __new__(cls, value):
        if not isinstance(value, int):
            raise ValueError("SmallInt must be an integer")
        elif not 0 < value < 2**16:
            raise ValueError("SmallInt out of bounds")
        return super().__new__(cls, value)

    def is_prime(self):
        return self in primes_16bit
```

We can define similar classes for other types.

Multiple and Predicative Dispatch

If we define these classes, we can simply use polymorphism. Each instance will use its preferred implementation of `is_prime()`.

```
nums = [
  SmallInt(64_489), SmallInt(64_487),
  MediumInt(262_147), MediumInt(262_143),
  BigInt(4_294_967_311), BigInt(4_294_967_309)
]
for num in nums:
    print(f"{num:,} is prime:",
            num.is_prime())
```

```
% python polymorphism.py
64,489 is prime: True
64,487 is prime: False
262,147 is prime: True
262,143 is prime: False
4,294,967,311 is prime: True
4,294,967,309 is prime: False
```

Multiple and Predicative Dispatch

Let try `gnosis-dispatch` after these other Python dispatch techniques:

```
from __future__ import annotations
from dispatch.dispatch import get_dispatcher
from primes import (
    is_small_prime,
    is_medium_prime,
    miller_rabin_prime,
    agrawal_kayal_saxena,
    gaussian_prime,
)
nums = get_dispatcher("nums")
```

Import implementations and create a *dispatcher.*

Multiple and Predicative Dispatch

A dispatcher is a namespace containing functions and implementations of functions.

```
@nums
def is_prime(n: int & 0 < n < 2**16) → bool:
    return is_small_prime(n)

@nums
def is_prime(n: 0 < n < 2**32) → bool:
    return is_medium_prime(n)

@nums(name="is_prime")
def mr_prime(n: int & n ≥ 2**32,
             confidence: float = 0.999_999):
    return miller_rabin_prime(n, confidence)
```

Annotations can have types and predicates.
Implementations can vary in arity.

Multiple and Predicative Dispatch

Let's add functions and implementations.

```
@nums(name="is_prime")
def aks_prime(n: int & n ≥ 2**32,
              confidence: float & confidence == 1.0):
    return agrawal_kayal_saxena(n)

# Gaussian prime is already annotated as ‘n: complex’
nums(name="is_prime")(gaussian_prime)

@nums
def is_twin_prime(n: int):
    "Check if n is part of a twin prime pair"
    return (nums.is_prime(n) and
      (nums.is_prime(n + 2) or nums.is_prime(n - 2)))
```

The decorator wraps an annotated function.

A second function has the same domain.

Multiple and Predicative Dispatch

Let's see what we've created:

```
>>> nums.describe()
nums bound implementations:
(0) is_prime
    n: int ∩ 0 < n < 2 ** 16
(1) is_prime
    n: Any ∩ 0 < n < 2 ** 32
(2) is_prime (re-bound 'mr_prime')
    n: int ∩ n ≥ 2 ** 32
    confidence: float ∩ True
(3) is_prime (re-bound 'aks_prime')
    n: int ∩ n ≥ 2 ** 32
    confidence: float ∩ confidence == 1.0
(4) is_prime (re-bound 'gaussian_prime')
    c: complex ∩ True
(0) is_twin_prime
    n: int ∩ True
```

Multiple and Predicative Dispatch

There might be a subtle weakness here.

```
(1) is_prime
    n: Any ∩ 0 < n < 2 ** 32
```

What if a user passes a float? Maybe add:

```
@nums
def is_prime(n: float):
    "Exclude floating point numbers"
    return False
```

We might instead only rely on the prior implementation doing the “right thing” for floating point numbers.

Multiple and Predicative Dispatch

Let's play around with our `nums` namespace:

```
>>> str(nums)
'nums with 2 functions bound to 6 implementations'

>>> nums.is_prime(64_489)   # True by direct search
True
>>> nums.is_prime(64_487)   # False by direct search
False
>>> nums.is_prime(262_147)  # True by trial division
True
>>> nums.is_prime(262_143)  # False by trial division
False

>>> is_small_prime(262_147) # Prime but not small
False
```

Multiple and Predicative Dispatch

Trial division can produce false positives (our implementation has finitely many divisors).

```
>>> for n in range(2**32, 2**32 + 500_000):
...     medium_prime = is_medium_prime(n)
...     is_prime = nums.is_prime(n)
...     if medium_prime != is_prime:
...         print(f"{n:,}: {medium_prime=} {is_prime=}")

4,295,098,369: medium_prime=True is_prime=False
4,295,229,443: medium_prime=True is_prime=False
4,295,360,521: medium_prime=True is_prime=False
```

`nums` selects the implementation for each call (*sort of* like polymorphism).

Multiple and Predicative Dispatch

Maybe with a whole different data type:

```
>>> nums.is_prime(-4 + 5j) # Gaussian primality
True
>>> nums.is_prime(+4 - 7j) # → False
```

Or different function in the namespace:

```
nums.is_twin_prime(617) # True (smaller of two)
nums.is_twin_prime(619) # True (larger of two)
nums.is_twin_prime(621) # False (not a prime)
nums.is_twin_prime(631) # False (not a twin)
```

Or an unsatisfiable argument:

```
>>> nums.is_prime(-1)
ValueError: No matching implementation for
args=(-1,), kws={}
```

Multiple and Predicative Dispatch

Trying a few more implementations.

```
>>> nums.is_prime(4_294_967_311) # Miller-Rabin
True
>>> nums.is_prime(4_294_967_309) # MR → False
>>> nums.is_prime(4_294_967_311, confidence=1.0) # AKS
True
>>> nums.is_prime(4_294_967_309, confidence=1.0) # AKS
False
```

Miller-Rabin is *very rarely* wrong, even at low confidences and known "liars."

```
>>> sum(nums.is_prime(4_295_038_231, confidence=0.01)
        for _ in range(1_000_000)) / 1_000_000
0.000105
```

Multiple and Predicative Dispatch

Toy code showed an equivalent task using the same name with differing data types, or satisfying different predicates.

`gnosis-dispatch` can aid development by allowing easier extensibility than traditional object-oriented programming.[1]

The initial `nations` example illustrates this.

[1]E.g. an actual mathematician could extend `nums` to add Eisenstein primes (complex numbers satisfying a predicate), Hurwitz primes (quaternion type), or other concepts I don't begin to understand.

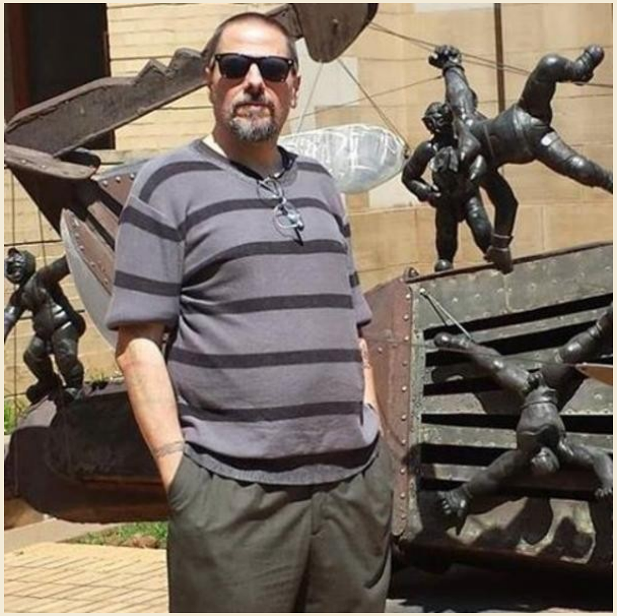
Multiple and Predicative Dispatch

(Coda) Expose custom types. E.g. `Authority` and `Place` were type annotations in `nations`.

```
from __future__ import annotations
from dispatch.dispatch import get_dispatcher
from world_data import Authority, Place
from data_tools import Analysis

nations = get_dispatcher(
    "nations", extra_types=[Authority, Place])

# Can add custom types and values as needed
def gt(a, b): return a > b

@nations(using=[Analysis, gt, {limit: 100}])
def new_function(arg: Analysis & gt(arg.val, limit)):
    #  ... code here ...
```

Multiple and Predicative Dispatch

Different dispatch abstractions than those usually taught in programming curricula can sometimes improve the extensibility and the structural clarity your programs.



david.mertz@seiu.org

pypi.org/project/gnosis-dispatch/

```
uv pip install
gnosis-dispatch
```

Questions?

PYCON AFRICA 25