Variadic Type Variables for Decorators and Tensors

Mark Mendoza & Vincent Siles

September ‘19 Python Typing Summit

Background

- Ivan proposed variadic generics last typing summit
- We encountered many decorators which fit into that target paradigm
  - We wanted to correctly type those decorators and begin to explore typing PyTorch
- We have implemented:
  - a prototype of `variadic=True` in Pyre & `pyre_extensions`
  - a new kind of type variable for return-type-modifying decorators
- We are here to get feedback on our approaches, and guidance from the community on how best to proceed

Outline

- **Parameter Specifications**
- List Variadics
- PyTorch Linear Regression
- Possible Extensions/Discussion

Parameter Specifications

Parameter Specifications: Motivation

- Many decorators are designed to accept both async and non-async functions, and add some async functionality to the function
- This means it takes a `Callable[[...], X]` and returns a `Callable[[...], Awaitable[X]]`
- We'd like to preserve all of the information about the parameters in this decoration
    - This includes names, `*args` and `**kwargs`, and default parameters

Parameter Specifications: Motivation

- Currently, to express the type of decorators that **modify the return type** of the decorated function, we can choose between:
  - Not modifying the signature at all
    - ```
      F = TypeVar("F", bound=Callable)
      def d(f: F) -> F:
      ```
  - Losing all signature information
    - ```
      def d(f: Callable[..., TReturn])-> Callable[..., TOther]:
      ```
  - Specializing the type accepted by the decorator
    - ```
      def d(f: Callable[[specific, types], TReturn])-> Callable[[specific, types], TOther]:
      ```
      ```

Parameter Specifications: Prototype

```python
from pyre_extensions import ParameterSpecification
from pyre_extensions.type_variable_operators import (
    PositionalArgumentsOf,
    KeywordArgumentsOf,
)

from typing import TypeVar, Callable, List


TParams = ParameterSpecification("TParams")
TReturn = TypeVar("TReturn")
```

Parameter Specifications: Prototype

```
def make_return_list(
    x: Callable[TParams, TReturn]
) -> Callable[TParams, List[TReturn]]:

    def decorated(
        *args: PositionalArgumentsOf[TParams],
        **kwargs: KeywordArgumentsOf[TParams]
    ) -> List[TReturn]:

        return [x(*args, **kwargs)]

    return decorated
```

Parameter Specifications: Prototype

```
def make_return_list(
    x: Callable[TParams, TReturn]
) -> Callable[TParams, List[TReturn]]: ...

@make_return_list
def foo(x: int, y: str, z: bool = False) -> int:
    return 12

foo

# `typing.Callable[
#   [Named(x, int), Named(y, str),Named(z, bool,default)],
#    typing.List[int]]`.
```

Outline

- Parameter Specifications
    - `TParams = ParameterSpecification("TParams")`
    - `type_variable_operators.PositionalArgumentsOf[TParams]`
    - `type_variable_operators.KeywordArgumentsOf[TParams]`
- **List Variadics**
- PyTorch Linear Regression
- Possible Extensions/Discussion

List Variadics

List Variadics: Motivation

- There are variadic functions that require a certain relationship between lists of types of arbitrary length
- **Parameter** transforming decorators
  - Some decorators modify parameters, not just return types
  - Parameter Specifications won’t be powerful enough
- Tensors are matrices of arbitrary dimension: intrinsically variadic

List Variadics: Prototype

```
from pyre_extensions import ListVariadic
Ts = ListVariadic("Ts")
```

List Variadics: Prototype

```python
from typing import Tuple, Callable
from pyre_extensions import ListVariadic
Ts = ListVariadic("Ts")


def duple(x: Tuple[Ts]) -> Tuple[Tuple[Ts], Tuple[Ts]]:
    return x, x


duple(
    (1, "A")
)  # => Tuple[Tuple[int, str], Tuple[int, str]]
```

List Variadics: Prototype

```
def identity(
    f: Callable[[Ts], TReturn]
) -> Callable[[Ts], TReturn]:
    return f


def foo(x: int, y: str) -> int:
    return x

foo # Callable[[Named(x, int), Named(y, str)], int]
identity(foo) # Callable[[int, str], int]
```

List Variadics: Prototype

```
def give_args_back(*args: Ts) -> Tuple[Ts]:
    return args

give_args_back(1, “A”, True)

   # typing.Tuple[int, str, bool]
```

Map Operator

Map Operator: Motivation

- Standard library functions like `map` and `asyncio.gather` are defined on arbitrary numbers of parameters which are **all wrapped in a parametric type**

```python
@overload
def map(__func: Callable[[_T1], _S], __iter1: Iterable[_T1]) -> Iterator[_S]: ...
@overload
def map(__func: Callable[[_T1, _T2], _S], __iter1: Iterable[_T1],
        __iter2: Iterable[_T2]) -> Iterator[_S]: ...
@overload
def map(__func: Callable[[_T1, _T2, _T3], _S],
        __iter1: Iterable[_T1],
        __iter2: Iterable[_T2],
        __iter3: Iterable[_T3]) -> Iterator[_S]: ...
@overload
def map(__func: Callable[[_T1, _T2, _T3, _T4], _S],
        __iter1: Iterable[_T1],
        __iter2: Iterable[_T2],
        __iter3: Iterable[_T3],
        __iter4: Iterable[_T4]) -> Iterator[_S]: ...
@overload
def map(__func: Callable[[_T1, _T2, _T3, _T4, _T5], _S],
        __iter1: Iterable[_T1],
        __iter2: Iterable[_T2],
        __iter3: Iterable[_T3],
        __iter4: Iterable[_T4],
        __iter5: Iterable[_T5]) -> Iterator[_S]: ...
@overload
def map(__func: Callable[..., _S],
        __iter1: Iterable[Any],
        __iter2: Iterable[Any],
        __iter3: Iterable[Any],
        __iter4: Iterable[Any],
        __iter5: Iterable[Any],
        __iter6: Iterable[Any],
        *iterables: Iterable[Any]) -> Iterator[_S]: ...
```

```python
@overload
def gather(
    coro_or_future1: _FutureT[_T1],
    *,
    loop: Optional[AbstractEventLoop] = ...,
    return_exceptions: bool = ...
) -> Future[Tuple[_T1]]:
    ...

@overload
def gather(
    coro_or_future1: _FutureT[_T1],
    coro_or_future2: _FutureT[_T2],
    *,
    loop: Optional[AbstractEventLoop] = ...,
    return_exceptions: bool = ...
) -> Future[Tuple[_T1, _T2]]:
    ...

@overload
def gather(
    coro_or_future1: _FutureT[_T1],
    coro_or_future2: _FutureT[_T2],
    coro_or_future3: _FutureT[_T3],
    *,
    loop: Optional[AbstractEventLoop] = ...,
    return_exceptions: bool = ...
) -> Future[Tuple[_T1, _T2, _T3]]:
    ...

@overload
def gather(
    coro_or_future1: _FutureT[_T1],
    coro_or_future2: _FutureT[_T2],
    coro_or_future3: _FutureT[_T3],
    coro_or_future4: _FutureT[_T4],
    *,
    loop: Optional[AbstractEventLoop] = ...,
    return_exceptions: bool = ...
) -> Future[Tuple[_T1, _T2, _T3, _T4]]:
    ...

@overload
def gather(
    coro_or_future1: _FutureT[_T1],
    coro_or_future2: _FutureT[_T2],
    coro_or_future3: _FutureT[_T3],
    coro_or_future4: _FutureT[_T4],
    coro_or_future5: _FutureT[_T5],
    *,
    loop: Optional[AbstractEventLoop] = ...,
    return_exceptions: bool = ...
) -> Future[Tuple[_T1, _T2, _T3, _T4, _T5]]:
    ...

@overload
def gather(
    coro_or_future1: _FutureT[Any],
    coro_or_future2: _FutureT[Any],
    coro_or_future3: _FutureT[Any],
    coro_or_future4: _FutureT[Any],
    coro_or_future5: _FutureT[Any],
    coro_or_future6: _FutureT[Any],
    *coros_or_futures: _FutureT[Any],
    loop: Optional[AbstractEventLoop] =
...,
    return_exceptions: bool = ...
) -> Future[Tuple[Any, ...]]:
    ...
```

Map Operator: Motivation

- Standard library functions like map and asyncio.gather are defined on arbitrary numbers of parameters which are **all wrapped in a parametric type**
  - map: `Iterable`
  - asyncio.gather: `_FutureT`
- Ivan proposed `GenericClass[Ts]` syntax for this situation

Map Operator: Prototype

- Leaning towards wordiness for now
- `pyre_extensions.type_variable_operators.Map[Iterable, Ts]` represents a list of types, `Iterable[Ts_0], Iterable[Ts_1], ..., Iterable[Ts_n]` where `Ts` is a `ListVariadic` that contains `Ts_0, Ts_1, ..., Ts_n`.

Map Operator: Prototype

```
def map(
    func: Callable[[Ts], TReturn],
    *args: Map[Iterable, Ts],
) -> TReturn: ...
```

```
map(takes_int, [1,2])                   # accepted
map(takes_int_str, [1,2], ["A", "B"])   # accepted
map(takes_int_str, ["A", "B"], [1, 2])  # rejected
```

Map Operator: Prototype

```
def asyncio.gather(
    *args: Map[Awaitable, Ts],
    loop: AbstractEventLoop = ...,
    return_exceptions: bool = ...
) -> Awaitable[Tuple[Ts]]: ...
```

Concatenate Operator

Concatenate Operator: Motivation

- We need to be able to move unary types on and off of a variadic
  - Ivan proposed the `Tuple[int, Expand[Ts]]` syntax
  - Necessary for the most common type of parameter modifying decorator
    - Adding or removing an argument

Concatenate Operator: Prototype

- Again, leaned towards wordiness for flexibility
- `pyre_extensions.type_variable_operators.Concatenate` works for arbitrary numbers of unaries before and after a `ListVariadic`
  - `Concatenate[int, str, Ts]`
  - `Concatenate[Ts, bool, float]`
  - `Concatenate[int, str, Ts, bool, float]`
  - ~~`Concatenate[Ts, Ts2]`~~

Concatenate Operator: Prototype

```
def prepend_addition_argument(
    f: Callable[[Ts], int]
) -> Callable[[Concatenate[int, Ts]], str]:
    def inner(x: int, *args: Ts) -> str:
        return str(x + f(*args))
    return inner


@prepend_addition_argument
def foo(x: int, y: int) -> int:
    return x * y

foo # Callable[[int, int, int], str]
```

Concatenate Operator: Prototype

```python
def simple_partial_application(
    f: Callable[[Concatenate[int, Ts]], TReturn]
) -> Callable[[Ts], TReturn]:
    def inner(*args: Ts) -> TReturn:
        return f(42, *args)
    return inner


@simple_partial_application
def foo(x: int, y: str, z: bool) -> float:
    return 3.5

foo # Callable[[str, bool], float]
```

Concatenate Operator: Prototype

- We can also ~~abuse~~ use this syntax to define classes with both normal and variadic parameters
    - Probably ultimately will need a “capture group” syntax

Concatenate Operator: Prototype

```
from pyre_extensions import Generic # typing.Generic does
                                    # arity validation

class Tensor(Generic[Concatenate[T, Ts]]):
    def el(self) -> T:
        ...

    def dims(self) -> Tuple[Ts]:
        ...
```

Outline

- Parameter Specifications
    - `TParams = ParameterSpecification("TParams")`
    - `type_variable_operators.PositionalArgumentsOf[TParams]`
    - `type_variable_operators.KeywordArgumentsOf[TParams]`
- List Variadics
    - `Ts = ListVariadic("Ts")`
    - `type_variable_operators.Map[ParametricClass, Ts]`
    - `type_variable_operators.Concatenate[unary, unary, Ts, unary]`
- **PyTorch Linear Regression**
- Possible Extensions/Discussion

PyTorch Linear Regression

[Original Example from PyTorch]

[Annotated Version]

Outline

- Parameter Specifications
  - `TParams = ParameterSpecification("TParams")`
  - `type_variable_operators.PositionalArgumentsOf[TParams]`
  - `type_variable_operators.KeywordArgumentsOf[TParams]`
- List Variadics
  - `Ts = ListVariadic("Ts")`
  - `type_variable_operators.Map[ParametricClass, Ts]`
  - `type_variable_operators.Concatenate[unary, unary, Ts, unary]`
- PyTorch Linear Regression
- **Possible Extensions/Discussion**

Discussion Topics

Discussion Outline

- Is this even worth it?
- Syntax
- Index operator
- Broadcasting
- IntVars
- Cat
- Sparse data structures
- Standardization
- Stubs

Discussion Topics

- Is this even worth it?
  - This is a pretty major complication to the Python type system
  - Implementing and specifying this will require a significant amount of engineering effort
  - Getting adoption from the ML community is not a sure thing
    - Maintainers need to be on board

Discussion Topics

- Why we think so:
  - There are returns at various levels of investment, not all or nothing
    - Minimal: better support of decorators and standard library functions
    - Moderate: cover most of numpy/pytorch, then add runtime validation
    - Complete: potentially transform dev experience of doing ML in python
  - We as type checker maintainers are ultimately the best positioned to implement this kind of analysis

Discussion Topics

- How should we adapt the syntax to be ergonomic without being ambiguous?
  - Trying to make “spreading” and mapping implicit at the same time can lead to ambiguity about what’s a true type and what’s actually another variadic entity
  - `Tuple[Ts]` would actually be a type (implicit spread)
  - `Iterable[Ts]` would not be a type (implicit map)
  - How would you actually map through a variadic type? In what circumstance is `Tuple[Ts]` equivalent to `Tuple[Ts_0], Tuple[Ts_1], ..., Tuple[Ts_n]`?
  - Trying to encode all of our operators into sugared shorthand could make new operators harder to add
- Index operator
  - `def access(t: Tuple[Ts], i: I) -> Index[Ts, I]`
  - `OutOfBounds` type?

Discussion Topics

- Broadcasting
  - NumPy defined semantics for how to handle when tensors have mismatched sizes
  - Apparently does what you meant to do *most of the time*
  - Many `Tensor` operations are implicitly broadcasted
  - Semantics are explicitly defined, but pretty complex

```
Two tensors are "broadcastable" if the following
rules hold:
  ●  Each tensor has at least one dimension.
  ●  When iterating over the dimension sizes,
     starting at the trailing dimension, the
     dimension sizes must either be equal, one
     of them is 1, or one of them does not
     exist.
If two tensors x, y are "broadcastable", the
resulting tensor size is calculated as follows:
  ●  If the number of dimensions of x and y are
     not equal, prepend 1 to the dimensions of
     the tensor with fewer dimensions to make
     them equal length.
  ●  Then, for each dimension size, the
     resulting dimension size is the max of the
     sizes of x and y along that dimension.
```

Discussion Topics

- What is the best way to implement/specify `IntVars`?
- How can we best represent operations like `cat`?
- How do we extend this to sparse data structures?
  - `ModelType`, `Key`
  - NamedTensors
  - Pandas dataframes
- How can we standardize the semantics here?
- How can we best cooperate on synthesizing stubs for these target libraries?

Outline

- Parameter Specifications
  - `TParams = ParameterSpecification("TParams")`
  - `type_variable_operators.PositionalArgumentsOf[TParams]`
  - `type_variable_operators.KeywordArgumentsOf[TParams]`
- List Variadics
  - `Ts = ListVariadic("Ts")`
  - `type_variable_operators.Map[ParametricClass, Ts]`
  - `type_variable_operators.Concatenate[unary, unary, Ts, unary]`
- PyTorch Linear Regression
- Possible Extensions/Discussion

Additional Notes

Current Implementation Approach

- Our overall strategy with type variables is **interval refinement**
    - `meet` new upper bound against existing one, `join` new lower bound against existing
- To extend this to new variable “kinds”, must define those operations on different domains
- We recursively break up complex types down towards single variables
- Intervals are refined at every new constraint (fail fast)
    - This is what this buys us vs. a unification approach
- The whole system is solved out at signature selection boundaries

Parameter Specifications for Abstract Classes

- Sometimes we'd like to define one method's signature based on that of another which is **abstract in the current class**.
- This comes up in PyTorch with `Module`
    - callable abstract base class
    - children are callable with same signature as they define for their `forward` method.
    - currently it has to be **stubbed**, as `def __call__(*args: object, **kwargs: object)`
- The following example is not yet implemented in Pyre

```python
class Module(Generic[TParams, TReturn]):
    @abstractmethod
    def forward(
        self,
        *args: PositionalArgumentsOf[TParams],
        **kwargs: KeywordArgumentsOf[TParams]
    ) -> TReturn:
        ...
```

```python
class Module(Generic[TParams, TReturn]):
    ...

    def __call__(
            self,
            *args: PositionalArgumentsOf[TParams],
            **kwargs: KeywordArgumentsOf[TParams]
     ) -> TReturn:

            # runs pre-hooks
            r = self.forward(*args, **kwargs)
            # runs post-hooks
            return r
```

```python
class SimpleModule(
    Module[
        ParameterSpecificationOf[SimpleModule.forward], int
    ]
):
    def forward(self, x: int, y: bool) -> int:
        return 7
```

```python
s = SimpleModule()

s.__call__
# `typing.Callable[[Named(x, int), Named(y, bool)], int]`.
```

PyTorch Linear Regression Excerpts

```python
Shape = ListVariadic("Shape")
DType = TypeVar("DType", int, float)

class Tensor(pyre_extensions.Generic[Concatenate[DType, Shape]]):
    ...
    # add a scalar to all cells of a tensor (simple broadcasting)
    @overload
    def __add__(
        self, other: DType
    ) -> "Tensor[Concatenate[DType, Shape]]":
        ...

    # Add two tensors of the exact same shape (no broadcasting)
    @overload
    def __add__(
        self, other: "Tensor[Concatenate[DType, Shape]]"
    ) -> "Tensor[Concatenate[DType, Shape]]":
        ...
```

```python
def mm(
    left: Tensor[DType, A, B], right: Tensor[DType, B, C]
) -> Tensor[DType, A, C]:
    ...


def randn(
    *args: Shape
) -> Tensor[Concatenate[float32, Shape]]:
    ...


# cannot express yet
def cat(l, n):
    ...
```

```python
class Linear(pyre_extensions.Generic[DType, D1, D2]):
    # F(X) = A * X + B
    # A is a two dimensions tensor (Matrix)
    # B is a one dimension tensor (Vector)
    # parameters returns the collection of A and B
    def parameters(self) -> Tuple[Tensor[DType, D2, D1], Tensor[DType, D2]]:
        …


N = TypeVar("N")

def smooth_l1_loss(
    refy: Tensor[DType, N, Literal[1]], y: Tensor[DType, N, Literal[1]]
) -> Tensor[DType, Literal[1]]: ...
```

```python
D1 = Literal[1]
D4 = Literal[4]
D32 = Literal[32]

W_target: Tensor[float32, D4, D1] = torch.randn(4, 1) * 5
b_target: Tensor[float32, D1] = torch.randn(1) * 5

N = TypeVar("N")


def make_features(x: Tensor[DType, N]) -> Tensor[DType, N, D4]:
    """Builds features i.e. a matrix with columns [x, x^2, x^3, x^4]."""
    x2 = torch.unsqueeze(x, 1)  # turns Tensor[N] into Tensor[N, 1]
    # We manually annotate the result of cat for now
    r: Tensor[DType, N, D4] = torch.cat([x2 ** i for i in range(1, 5)], 1)
    return r
```

```python
def f(x: Tensor[float32, N, D4]) -> Tensor[float32, N, D1]:
    """Approximated function."""
    return torch.mm(x, W_target) + b_target.item()


I = IntVar("I")


def get_batch(batch_size: I) -> Tuple[Tensor[float32, I, D4], Tensor[float32, I,
D1]]:
    """Builds a batch i.e. (x, f(x)) pair."""
    random: Tensor[float32, D32] = torch.randn(batch_size)
    x = make_features(random)
    y = f(x)
    return x, y
```

```python
fc: torch.nn.Linear[float32, D4, D1] = torch.nn.Linear(W_target.size(0), 1)


for batch_idx in count(1):
    ...

    batch_x, batch_y = get_batch()

    output = F.smooth_l1_loss(fc(batch_x), batch_y)

    param1, param2 = fc.parameters()

    # both additions are checked to have matching dimensions
    param1.data.add_(-0.1 * param1.grad.data)
    param2.data.add_(-0.1 * param2.grad.data)

    ...
```

```python
def poly_desc(W: Sequence[T], b: Tensor[DType, D1]) -> str:
    """Creates a string description of a polynomial."""
    result = "y = "
    for i, w in enumerate(W):
        result += "{:+.2f} x^{} ".format(w, len(W) - i)
    result += "{:+.2f}".format(b[0])
    return result

print("Loss: {:.6f} after {} batches".format(loss, final_index))
print("==> Learned function:\t" + poly_desc(fc.weight.view(-1), fc.bias))
print("==> Actual function:\t" + poly_desc(W_target.view(-1), b_target))
```