A Deep Dive into Beam Python Type Hinting

BEAM SUMMIT
NYC 2025

Jack R. McCluskey

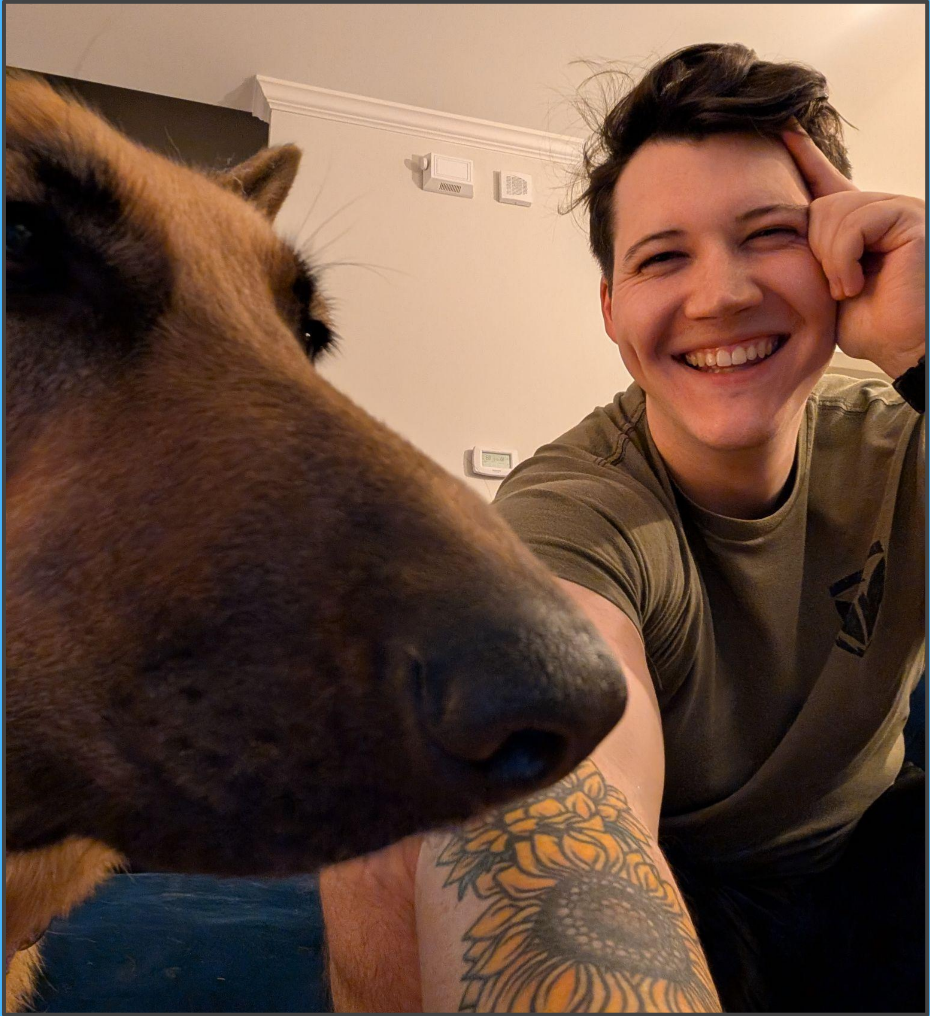
SWE @ Google

Agenda

- Background
  - Static vs. Runtime Type Checking
  - Type Compatibility
  - Why Types Matter to Beam
- The Infrastructure
  - Applying Type Hints
  - Internal Type Representations
  - Processing and Compatibility
- “Trivial” Inference

About Me

- Graduated from UNC Chapel Hill in 2020
- Joined Google in July of 2020 through the Engineering Residency Program
- Started working on Google Cloud Dataflow in June of 2021



Static Type Checking

- Static analysis of written code, usually in an IDE
- Sanity checks usage of an object based on its hinted type
- Does not necessarily reflect realities at runtime

Runtime Type Checking

- Programmatically checking type compatibility during code execution
- Useful when path of inputs/outputs is not necessarily static (like in a Beam pipeline)
- Still not necessarily reflective of actual types at execution

Type Compatibility

- Python is duck-typed
- Instead of direct inheritance, compatibility is based on a subtype relationship
- A type is considered a subtype of a parent type **if**:
  - The subtype has valid attribute values for the parent type
  - The subtype implements the methods of the parent type
- Special case: the Any type is compatible with *every* type and *every* type is compatible with Any

Why Types Matter to Beam

- Static checking gives us sanity checks within DoFn definitions
  - e.g. assuming the input PCollection is of the type you expect, your IDE will warn you if you try to access something on an object in a way that isn't guaranteed to be there
- Runtime checking gives us sanity checks *between* transforms in a pipeline graph
- Beam Python also uses type hints to select more efficient coders

Applying Type Hints

- Beam Python allows transforms to be type annotated in a few different ways:
    - PEP 484 Function Annotations
    - Method Chaining in Pipeline Definitions
    - Decorators

Function Annotation

```
class WordExtractingDoFn(beam.DoFn):
  def process(self, element: str) -> str:
    return re.findall(r'[\w\']+', element, re.UNICODE)
```

Method Chaining

```
class WordExtractingDoFn(beam.DoFn):
  def process(self, element):
    return re.findall(r'[\w\']+', element, re.UNICODE)

...

counts = (
      lines
      | 'Split' >> (beam.ParDo(WordExtractingDoFn()).with_output_types(str))
      | 'PairWithOne' >> beam.Map(lambda x: (x, 1))
      | 'GroupAndSum' >> beam.CombinePerKey(sum)
```

Function Decorators

```
@typehints.with_input_types(str)
@typehints.with_output_types(str)
class WordExtractingDoFn(beam.DoFn):
  def process(self, element):
    return re.findall(r'[\w\']+', element, re.UNICODE)
```

Internal Type Representations

- Beam uses internal type representations of common container types rather than the Python-native versions
- Representations inherit from one of two base classes
  - TypeConstraint
    - Implement a _consistent_with_check() method that contains logic for determining if a given type is consistent with the constraint
  - CompositeTypeHint
    - Function as a TypeConstraint factory for more complex type representations
    - Accept types via __getitem__() and parameterizes a TypeConstraint with that argument
    - Oftentimes contains a nested definition of a TypeConstraint for specific use

Internal Type Representations

```
class DictConstraint(TypeConstraint):
    def __init__(self, key_type, value_type):
      self.key_type = normalize(key_type)
      self.value_type = normalize(value_type)

    def _consistent_with_check_(self, sub):
          return (
              isinstance(sub, self.__class__) and
              is_consistent_with(sub.key_type, self.key_type) and
              is_consistent_with(sub.value_type, self.value_type))
```

Internal Type Representations

```
class DictHint(CompositeTypeHint):
  def __getitem__(self, type_params):
    # Type param must be a (k, v) pair.
    if not isinstance(type_params, tuple):
      raise TypeError(
          'Parameter to Dict type-hint must be a tuple of types: '
          'Dict[.., ..].')

    if len(type_params) != 2:
      raise TypeError(
          'Length of parameters to a Dict type-hint must be exactly 2. Passed '
          'parameters: %s, have a length of %s.' %
          (type_params, len(type_params)))

    key_type, value_type = type_params

    ...

    return self.DictConstraint(key_type, value_type)
```

Processing Types to Internal Versions

```python
# Example Matcher
def _match_is_dict(user_type):
  return _is_primitive(user_type, dict) or _safe_issubclass(user_type, dict)

# Example TypeMapEntry
_TypeMapEntry(match=_match_is_dict, arity=2, beam_type=typehints.Dict)
```

Processing Types to Internal Versions

```
# Find the first matching entry.
matched_entry = next((entry for entry in type_map if entry.match(typ)), None)
if not matched_entry:
    _LOGGER.info('Using Any for unsupported type: %s', typ)
    return typehints.Any
```

Processing Types to Internal Versions

- We process known container types into Beam-internal representations in native_type_compatibility.py
  - Built-ins, typing module types, collections and collections.abc types are “known” container types
- A list of named tuples defines the order in which types are checked, a matcher function to check against a type, and the arity of that type
  - Order in this list *does* matter, we want to check against specific categories first then broaden later (e.g. a frozenset is also a set, so we should check if a type is a frozenset first)

Checking Compatibility

- The actual type checking function is the is_consistent_with() function
- Conceptually, we leverage as much information as we can before falling back to Python's built-in issubclass() check

Checking Compatibility

1. **Check Common Cases**
   Exact matches and integer relationships
2. **Normalize**
   Coerce known types to Beam-internal types, leave everything else as-is
3. **Check Against TypeConstraints**
   Use the defined Beam-internal compatibility functions to check types
4. **Fall Back to issubclass()**
   If we don't have any TypeConstraints, pass to issubclass() for the final check

Checking Compatibility

```
def is_consistent_with(sub, base):
  if sub == base:
    return True
  if sub is int and base in (float, complex):
    return True
  if sub is float and base is complex:
    return True
  sub = normalize(sub, none_as_type=True)
  base = normalize(base, none_as_type=True)
```

Checking Compatibility

```
def normalize(x, none_as_type=False):
  if none_as_type and x is None:
    return type(None)
  # Convert bare builtin types to correct type hints directly
  elif x in _KNOWN_PRIMITIVE_TYPES:
    return _KNOWN_PRIMITIVE_TYPES[x]
  elif getattr(x, '__module__',
               None) in ('typing', 'collections', 'collections.abc') or getattr(
                   x, '__origin__', None) in _KNOWN_PRIMITIVE_TYPES:
    beam_type = native_type_compatibility.convert_to_beam_type(x)
    if beam_type != x:
      # We were able to do the conversion.
      return beam_type
    else:
      # It might be a compatible type we don't understand.
      return Any
  return x
```

Checking Compatibility

```
if isinstance(sub, AnyTypeConstraint) or isinstance(base, AnyTypeConstraint):
    return True
  elif isinstance(sub, UnionConstraint):
    return all(is_consistent_with(c, base) for c in sub.union_types)
  elif isinstance(base, TypeConstraint):
    return base._consistent_with_check_(sub)
  elif isinstance(sub, RowTypeConstraint):
    return base == Row
  elif isinstance(sub, TypeConstraint):
    # Nothing but object lives above any type constraints.
    return base == object
  elif getattr(base, '__module__', None) == 're':
    return regex_consistency(sub, base)
  elif is_typing_generic(base):
    return False
  return issubclass(sub, base)
```

“Trivial” Inference

- “Trivial” is a misnomer, as the trivial inference code in Beam Python is actually a CPython emulator
- The code emulates the underlying CPython byte code for any untyped function (but most importantly lambdas) to determine possible return types

CPython Operations

- CPython operations are similar to any sort of machine-level code
- Each instruction has an opcode, instruction size, and potential arguments for each.
- For each opcode, interactions with the stack and potential branches must be emulated with respect to types of potential values, but not the values themselves

The Emulation Loop

- Create initial objects to track emulated state
  - FrameState object for things like the stack, keyword names, etc.
  - Sets of returned and yielded types
- Extract the bytecode from the function using the dis package
  - Organize the individual instructions into a dictionary, with the byte offset as the key and the actual instruction as the value
- Start to loop over the program, emulating each instruction that directly impacts the state we track
  - Some operations like CACHEs are no-ops for our purposes

Branching

- Some CPython operations introduce logical branches
  - JMP_IF_TRUE, POP_IF_FALSE, etc.
- To cover varying return types from these branches, we have to evaluate both branches
  - Deep copy the current state of the frame (the “do not branch” case) and save it in a list of states
  - Modify the active frame based on the operation (the “branch” case)
  - Emulate the active frame until completion
  - Retrieve the saved state and resume emulation

Further Reading

- Doc overview of this content with code references
    - https://s.apache.org/beam-python-type-hinting-overview

Jack R. McCluskey

QUESTIONS?

jrmccluskey@apache.org
@jrmccluskey.com on BlueSky
jrmccluskey on Linkedin
jrmccluskey on Github