```
import _ast
import ast
import asyncio
import itertools
import json
import logging
import typing

from collections import defaultdict, Counter
from types import ModuleType, FunctionType

import numpy as np

import node_textual_representation_singleton
import switches_singleton
import typing_constraints_singleton
from breadth_first_search_layers import breadth_first_search_layers
from determine_number_of_type_variables import determine_number_of_type_variables
from disjoint_set import DisjointSet
from get_attributes_in_runtime_class import get_attributes_in_runtime_class
from relations import NonEquivalenceRelationGraph, NonEquivalenceRelationTuple, NonEquivalenceRelationType
from runtime_term import RuntimeTerm, Instance
from type_ascription import type_ascription
from type_definitions import RuntimeClass
from class_query_database import ClassQueryDatabase
from typeshed_client_ex.client import Client

from typeshed_client_ex.type_definitions import TypeshedTypeAnnotation, TypeshedClass, from_runtime_class, \
    TypeshedTypeVariable, subscribe, replace_type_variables_in_type_annotation, Subscription


def dump_confidence_and_possible_class_list(
        confidence_and_possible_class_list: list[tuple[float, TypeshedClass]],
        class_inference_log_file_io: typing.IO
):
    confidence_and_possible_class_string_list_list: list[list[float, str]] = [
        [confidence, str(possible_class)]
        for confidence, possible_class in confidence_and_possible_class_list
    ]

    json.dump(confidence_and_possible_class_string_list_list, class_inference_log_file_io)
    class_inference_log_file_io.write('\n')


class TypeInference:
    def __init__(
            self,
            type_query_database: ClassQueryDatabase,
```

```
49:             client: Client
50:     ):
51:         self.type_query_database = type_query_database
52:         self.client = client
53:
54:         self.class_inference_cache: dict[
55:             frozenset[ast.AST],
56:             tuple[
57:                 list[tuple[float, TypeshedTypeAnnotation]],
58:                 bool
59:             ]
60:         ] = dict()
61:         self.type_inference_cache: dict[frozenset[ast.AST], TypeshedTypeAnnotation] = dict()
62:
63:     def infer_classes_for_nodes(
64:             self,
65:             nodes: frozenset[ast.AST],
66:             indent_level: int = 0,
67:             cosine_similarity_threshold: float = 1e-1
68:     ) -> tuple[
69:         list[tuple[float, TypeshedClass]],  # class inference confidences and classs
70:         bool  # whether runtime class can be instance-of types.NoneType
71:     ]:
72:         indent = '    ' * indent_level
73:
74:         # Has a record in cache
75:         if nodes in self.class_inference_cache:
76:             logging.info(
77:                 '%sCache hit when performing class inference for %s.',
78:                 indent,
79:                 nodes
80:             )
81:
82:             return self.class_inference_cache[nodes]
83:         else:
84:             # No record in cache
85:             logging.info(
86:                 '%sCache miss when performing class inference for %s.',
87:                 indent,
88:                 nodes
89:             )
90:
91:             # Determine whether it can be None.
92:
93:             can_be_none: bool = False
94:             non_none_instance_classes: set[RuntimeClass] = set()
95:
96:             runtime_term_sharing_node_disjoint_set_top_node_set = {
```

```
 97:                 typing_constraints_singleton.runtime_term_sharing_node_disjoint_set.find(node)
 98:                 for node in nodes
 99:             }
100:
101:             for runtime_term_sharing_node_disjoint_set_top_node in runtime_term_sharing_node_disjoint_set_top_n
102:                 runtime_term_set = \
103:                 typing_constraints_singleton.runtime_term_sharing_equivalent_set_top_nodes_to_runtime_term_sets
104:                     runtime_term_sharing_node_disjoint_set_top_node
105:                 ]
106:                 for runtime_term in runtime_term_set:
107:                     if isinstance(runtime_term, Instance):
108:                         instance_class = runtime_term.class_
109:                         if isinstance(instance_class, type(None)):
110:                             can_be_none = True
111:                         else:
112:                             non_none_instance_classes.add(instance_class)
113:
114:             logging.info(
115:                 '%sCan %s be None? %s',
116:                 indent,
117:                 nodes, can_be_none
118:             )
119:
120:             # Initialize aggregate attribute counter.
121:             aggregate_attribute_counter: Counter[str] = Counter()
122:
123:             for breadth_first_search_layer in breadth_first_search_layers(
124:                     typing_constraints_singleton.node_containment_graph,
125:                     nodes
126:             ):
127:                 logging.info(
128:                     '%sCurrent breadth-first-search layer: %s', indent, breadth_first_search_layer
129:                 )
130:
131:                 for node in breadth_first_search_layer:
132:                     attribute_counter = typing_constraints_singleton.nodes_to_attribute_counters[
133:                         node
134:                     ]
135:
136:                     logging.info(
137:                         '%sAttribute counter for %s: %s',
138:                         indent,
139:                         node_textual_representation_singleton.node_to_textual_representation_dict.get(
140:                             node,
141:                             str(node)
142:                         ),
143:                         attribute_counter
144:                     )
```

```
145:
146:                 aggregate_attribute_counter.update(attribute_counter)
147:
148:             logging.info(
149:                 '%sAggregate attribute counter for %s: %s',
150:                 indent,
151:                 nodes,
152:                 aggregate_attribute_counter
153:             )
154:
155:             # Query possible classes.
156:
157:             confidence_and_possible_class_list: list[tuple[float, TypeshedClass]] = list()
158:
159:             if (
160:                     switches_singleton.shortcut_single_class_covering_all_attributes
161:                     and len(non_none_instance_classes) == 1
162:                     and set(aggregate_attribute_counter.keys()).issubset(
163:                         get_attributes_in_runtime_class(next(iter(non_none_instance_classes)))
164:                     )
165:             ):
166:
167:                 single_instance_class_covering_all_attributes = next(iter(non_none_instance_classes))
168:                 confidence_and_possible_class_list.append(
169:                     (1, from_runtime_class(single_instance_class_covering_all_attributes))
170:                 )
171:             else:
172:                 (
173:                     possible_class_ndarray,
174:                     cosine_similarity_ndarray,
175:                 ) = self.type_query_database.query(aggregate_attribute_counter)
176:
177:                 nonzero_cosine_similarity_indices = (cosine_similarity_ndarray > cosine_similarity_threshold)
178:
179:                 selected_possible_class_ndarray = possible_class_ndarray[nonzero_cosine_similarity_indices]
180:                 selected_cosine_similarity_ndarray = cosine_similarity_ndarray[nonzero_cosine_similarity_indice
181:
182:                 argsort = np.argsort(selected_cosine_similarity_ndarray)
183:
184:                 for i in argsort[-1::-1]:
185:                     possible_class = selected_possible_class_ndarray[i]
186:                     cosine_similarity = float(selected_cosine_similarity_ndarray[i])
187:
188:                     confidence_and_possible_class_list.append(
189:                         (cosine_similarity, possible_class)
190:                     )
191:
192:                 logging.info(
```

```
193:                     '%sPossible types queried for %s based on attributes: %s',
194:                     indent,
195:                     nodes,
196:                     confidence_and_possible_class_list
197:                 )
198:
199:             return_value = confidence_and_possible_class_list, can_be_none
200:
201:             self.class_inference_cache[nodes] = return_value
202:
203:             return return_value
204:
205:     def infer_type(
206:             self,
207:             node_set: frozenset[ast.AST],
208:             depth: int = 0,
209:             cosine_similarity_threshold: float = 1e-1,
210:             depth_limit: int = 3,
211:             first_level_class_inference_failed_fallback: TypeshedClass = TypeshedClass('typing', 'Any'),
212:             non_first_level_class_inference_failed_fallback: TypeshedClass = TypeshedClass('typing', 'Any'),
213:             class_inference_log_file_io: typing.Optional[typing.IO] = None
214:     ) -> TypeshedTypeAnnotation:
215:         indent = '    ' * depth
216:
217:         # Has a record in cache
218:         if node_set in self.type_inference_cache:
219:             logging.info(
220:                 '%sCache hit when performing type inference for %s.',
221:                 indent,
222:                 node_set
223:             )
224:
225:             return self.type_inference_cache[node_set]
226:         else:
227:             # No record in cache
228:             logging.info(
229:                 '%sCache miss when performing type inference for %s.',
230:                 indent,
231:                 node_set
232:             )
233:
234:             if depth > depth_limit:
235:                 logging.error(
236:                     '%sRecursive type inference exceeded depth limit of %s. Returning %s.',
237:                     indent,
238:                     depth_limit,
239:                     non_first_level_class_inference_failed_fallback
240:                 )
```

```
241:
242:                 return_value = non_first_level_class_inference_failed_fallback
243:             else:
244:                 # Part 1: Infer possible classes.
245:                 logging.info(
246:                     '%sPerforming class inference for %s.',
247:                     indent,
248:                     node_set
249:                 )
250:
251:                 (
252:                     confidence_and_possible_class_list,
253:                     can_be_none
254:                 ) = self.infer_classes_for_nodes(
255:                     node_set,
256:                     depth + 1,
257:                     cosine_similarity_threshold
258:                 )
259:
260:                 if class_inference_log_file_io is not None:
261:                     dump_confidence_and_possible_class_list(
262:                         confidence_and_possible_class_list,
263:                         class_inference_log_file_io
264:                     )
265:
266:                 # Part 2: Infer type variables for possible classes to get final type inference results.
267:                 if not confidence_and_possible_class_list:
268:                     top_class_prediction = (
269:                         first_level_class_inference_failed_fallback
270:                         if depth == 0
271:                         else non_first_level_class_inference_failed_fallback
272:                     )
273:
274:                     logging.info(
275:                         '%sNo possible classes queried for %s based on attributes. Using %s.',
276:                         indent,
277:                         node_set,
278:                         top_class_prediction
279:                     )
280:                 else:
281:                     (
282:                         top_class_prediction_confidence,
283:                         top_class_prediction
284:                     ) = confidence_and_possible_class_list[0]
285:
286:                     logging.info(
287:                         '%sTop class prediction: %s',
288:                         indent,
```

```
289:                         top_class_prediction
290:                     )
291:
292:                 return_value = top_class_prediction
293:
294:             self.type_inference_cache[node_set] = return_value
295:
296:             return return_value
```