```
 1: import _ast
 2: import ast
 3: import builtins
 4: import collections.abc
 5: import itertools
 6: import logging
 7: import typing
 8: from collections import defaultdict
 9:
10: import debugging_singleton
11: import switches_singleton
12: from definitions_to_runtime_terms_mappings_singleton import top_level_class_definitions_to_runtime_classes, unw
13: from get_attributes_in_runtime_class import get_non_dynamic_attributes_in_runtime_class
14: from get_dict_for_runtime_class import get_dict_for_runtime_class
15: from get_parameters import get_parameters
16: from module_names_to_imported_names_to_runtime_objects_singleton import module_names_to_imported_names_to_runti
17: from parameter_lists_and_symbolic_return_values_singleton import nodes_to_parameter_lists_parameter_name_to_par
18: from relations import NonEquivalenceRelationType
19: from typing_constraints_singleton import create_new_node, add_runtime_terms, set_node_to_be_instance_of, set_eq
20:     add_relation, create_related_node, update_attributes, \
21:     add_argument_of_returned_value_of_relations, add_containment_relation, add_two_way_containment_relation
22: from runtime_term import *
23: from node_visitor import *
24: from type_definitions import *
25: from unwrap import unwrap
26:
27:
28: unaryop_to_attribute: dict[type, str] = {
29:     ast.Invert: '__invert__',
30:     ast.UAdd: '__pos__',
31:     ast.USub: '__neg__'
32: }
33:
34: operator_to_attribute: dict[type, str] = {
35:     ast.Add: '__add__',
36:     ast.Sub: '__sub__',
37:     ast.Mult: '__mul__',
38:     ast.MatMult: '__matmul__',
39:     ast.Div: '__truediv__',
40:     ast.Mod: '__mod__',
41:     ast.Pow: '__pow__',
42:     ast.LShift: '__lshift__',
43:     ast.RShift: '__rshift__',
44:     ast.BitOr: '__or__',
45:     ast.BitXor: '__xor__',
46:     ast.BitAnd: '__and__',
47:     ast.FloorDiv: '__floordiv__'
48: }
```

```
49:
50: cmpop_to_attribute: dict[type, str] = {
51:     ast.Eq: '__eq__',
52:     ast.NotEq: '__ne__',
53:     ast.Lt: '__lt__',
54:     ast.LtE: '__le__',
55:     ast.Gt: '__gt__',
56:     ast.GtE: '__ge__'
57: }
58:
59:
60: async def collect_and_resolve_typing_constraints(
61:         module_names_to_module_nodes: typing.Mapping[str, ast.Module]
62: ):
63:     """
64:     Collect and resolve typing constraints based on the semantics of each AST node.
65:     """
66:     # ------------------------------------------------------------------------------------------ #
67:     # For the unwrapped runtime functions and runtime classes,
68:     # And the literals True, False, Ellipsis, None, NotImplemented in builtins,
69:     # And each imported name within a module,
70:     # Initialize nodes which 'define' them,
71:     # And associate them with adequate runtime values.
72:     module_names_to_names_to_dummy_ast_nodes: defaultdict[str, dict[str, ast.AST]] = defaultdict(dict)
73:
74:     names_to_nodes_for_builtins: dict[str, _ast.AST] = dict()
75:
76:     for key, value in builtins.__dict__.items():
77:         if isinstance(value, (UnwrappedRuntimeFunction, RuntimeClass)):
78:             node = await create_new_node()
79:
80:             names_to_nodes_for_builtins[key] = node
81:             await add_runtime_terms(node, {value})
82:
83:     for value in (True, False, Ellipsis, None, NotImplemented):
84:         key = str(value)
85:         node = await create_new_node()
86:
87:         names_to_nodes_for_builtins[key] = node
88:         await set_node_to_be_instance_of(node, type(value))
89:
90:     for module_name, imported_names_to_runtime_objects in module_names_to_imported_names_to_runtime_objects.ite
91:         module_names_to_names_to_dummy_ast_nodes[module_name].update(names_to_nodes_for_builtins)
92:
93:         for imported_name, runtime_object in imported_names_to_runtime_objects.items():
94:             node = await create_new_node()
95:             module_names_to_names_to_dummy_ast_nodes[module_name][imported_name] = node
96:
```

```
            unwrapped_runtime_object = unwrap(runtime_object)
            runtime_term: RuntimeTerm | None = None

            if isinstance(unwrapped_runtime_object, Module):
                runtime_term = unwrapped_runtime_object
            if isinstance(unwrapped_runtime_object, RuntimeClass):
                runtime_term = unwrapped_runtime_object
            if isinstance(unwrapped_runtime_object, UnwrappedRuntimeFunction):
                if unwrapped_runtime_object in unwrapped_runtime_functions_to_named_function_definitions:
                    runtime_term = unwrapped_runtime_functions_to_named_function_definitions[unwrapped_runtime_
                else:
                    runtime_term = unwrapped_runtime_object

            if runtime_term is not None:
                logging.info(
                    'Matched imported name %s in module %s with unwrapped runtime object %s to runtime term %s'
                    imported_name, module_name, unwrapped_runtime_object, runtime_term
                )
                await add_runtime_terms(node, {runtime_term})

            else:
                logging.error(
                    'Cannot match imported name %s in module %s with unwrapped runtime object %s to a runtime t
                    imported_name, module_name, unwrapped_runtime_object
                )

    # ------------------------------------------------------------------------------------------ #
    # Update the runtime term sets of class definitions and function definitions.
    async def update_runtime_term_sets_callback(
            scope_stack: list[NodeProvidingScope],
            class_stack: list[ast.ClassDef],
            node: _ast.AST
    ):
        if isinstance(node, (ast.FunctionDef, ast.AsyncFunctionDef, ast.Lambda)):
            await add_runtime_terms(node, {node})
        # ast.ClassDef(name, bases, keywords, starargs, kwargs, body, decorator_list)
        if isinstance(node, ast.ClassDef):
            # Update the runtime term set of the current type variable.
            if node in top_level_class_definitions_to_runtime_classes:
                runtime_class = top_level_class_definitions_to_runtime_classes[node]
                await add_runtime_terms(node, {runtime_class})

    for module_name, module_node in module_names_to_module_nodes.items():
        await AsyncScopedNodeVisitor(update_runtime_term_sets_callback).visit(module_node)

    # ------------------------------------------------------------------------------------------ #
    # Set the default values of parameters to be equivalent to the corresponding type variables.
    async def handle_parameter_default_values_callback(
```

```
            scope_stack: list[NodeProvidingScope],
            class_stack: list[ast.ClassDef],
            node: _ast.AST
    ):
        # ast.FunctionDef(name, args, body, decorator_list, returns, type_comment)
        # ast.AsyncFunctionDef(name, args, body, decorator_list, returns, type_comment)
        # ast.Lambda(args, body)
        if isinstance(node, (ast.FunctionDef, ast.AsyncFunctionDef, ast.Lambda)):
            # Get the parameter list of the current function.
            posargs, _, kwonlyargs, _ = get_parameters(node)

            # N posargs_defaults align with the *last* N posargs
            # N kw_defaults align with N kwonlyargs (though they may be None's)
            posargs_defaults = node.args.defaults
            kwonlyargs_defaults = node.args.kw_defaults

            for posarg, posarg_default in zip(
                reversed(posargs),
                reversed(posargs_defaults)
            ):
                await add_containment_relation(
                    superset_node=posarg,
                    subset_node=posarg_default
                )

            for kwonlyarg, kwonlyarg_default in zip(
                kwonlyargs,
                kwonlyargs_defaults
            ):
                if kwonlyarg_default is not None:
                    await add_containment_relation(
                        superset_node=kwonlyarg,
                        subset_node=kwonlyarg_default
                    )

    if switches_singleton.handle_parameter_default_values:
        for module_name, module_node in module_names_to_module_nodes.items():
            await AsyncScopedNodeVisitor(handle_parameter_default_values_callback).visit(module_node)

    # ------------------------------------------------------------------------------------------ #
    # Collect return value information for functions.
    # Resolve the (real) return value sets of nodes providing scope.
    nodes_providing_scope_to_apparent_return_value_sets: dict[NodeProvidingScope, set[ast.AST]] = dict()
    nodes_providing_scope_to_yield_value_sets: dict[NodeProvidingScope, set[ast.AST]] = dict()
    nodes_providing_scope_to_send_value_sets: dict[NodeProvidingScope, set[ast.AST]] = dict()
    nodes_providing_scope_returning_generators: set[NodeProvidingScope] = set()
    nodes_providing_scope_returning_coroutines: set[NodeProvidingScope] = set()

```

```
193:     async def collect_parameter_return_value_information_callback(
194:             scope_stack: list[NodeProvidingScope],
195:             class_stack: list[ast.ClassDef],
196:             node: _ast.AST
197:     ):
198:         # ast.FunctionDef(name, args, body, decorator_list, returns, type_comment)
199:         # ast.AsyncFunctionDef(name, args, body, decorator_list, returns, type_comment)
200:         if isinstance(node, (ast.FunctionDef, ast.AsyncFunctionDef)):
201:             # Initialize return type set, yield value set, send value set of the current scope.
202:             nodes_providing_scope_to_apparent_return_value_sets[node] = set()
203:             nodes_providing_scope_to_yield_value_sets[node] = set()
204:             nodes_providing_scope_to_send_value_sets[node] = set()
205:
206:             if isinstance(node, ast.AsyncFunctionDef):
207:                 nodes_providing_scope_returning_coroutines.add(node)
208:         # ast.Lambda(args, body)
209:         if isinstance(node, ast.Lambda):
210:             # Initialize return type set, yield value set, send value set of the current scope.
211:             nodes_providing_scope_to_apparent_return_value_sets[node] = {node.body}
212:             nodes_providing_scope_to_yield_value_sets[node] = set()
213:             nodes_providing_scope_to_send_value_sets[node] = set()
214:         # ast.Return(value)
215:         if isinstance(node, ast.Return):
216:             if scope_stack:
217:                 current_scope = scope_stack[-1]
218:
219:                 if node.value is not None:
220:                     # Add the type variable of `value` to the return type set of the current scope.
221:                     nodes_providing_scope_to_apparent_return_value_sets[current_scope].add(node.value)
222:             else:
223:                 logging.error('Cannot handle ast.Return outside of a scope!')
224:         # ast.Yield(value)
225:         if isinstance(node, ast.Yield):
226:             if scope_stack:
227:                 current_scope = scope_stack[-1]
228:
229:                 nodes_providing_scope_returning_generators.add(current_scope)
230:
231:                 if node.value is not None:
232:                     # Add the type variable of `value` to the yield type set of the current scope.
233:                     nodes_providing_scope_to_yield_value_sets[current_scope].add(node.value)
234:
235:                 # Add the current type variable to the send type set of the current scope.
236:                 nodes_providing_scope_to_send_value_sets[current_scope].add(node)
237:             else:
238:                 logging.error('Cannot handle ast.Yield outside of a scope!')
239:         # ast.YieldFrom(value)
240:         if isinstance(node, ast.YieldFrom):
```

```
241:             if scope_stack:
242:                 current_scope = scope_stack[-1]
243:
244:                 nodes_providing_scope_returning_generators.add(current_scope)
245:
246:                 # Add the $IterTargetOf$ the type variable of `value` to the yield type set of the current scop
247:                 nodes_providing_scope_to_yield_value_sets[current_scope].add(
248:                     await create_related_node(node.value, NonEquivalenceRelationType.IterTargetOf)
249:                 )
250:
251:                 # Add the $SendTargetOf$ the type variable of `value` to the send type set of the current scope
252:                 nodes_providing_scope_to_send_value_sets[current_scope].add(
253:                     await create_related_node(node.value, NonEquivalenceRelationType.SendTargetOf)
254:                 )
255:             else:
256:                 logging.error('Cannot handle ast.YieldFrom outside of a scope!')
257:
258:     for module_name, module_node in module_names_to_module_nodes.items():
259:         await AsyncScopedNodeVisitor(collect_parameter_return_value_information_callback).visit(module_node)
260:
261:     nodes_providing_scope_set = set().union(
262:         nodes_providing_scope_to_apparent_return_value_sets.keys(),
263:         nodes_providing_scope_to_yield_value_sets.keys(),
264:         nodes_providing_scope_to_send_value_sets.keys()
265:     ) & nodes_to_parameter_lists_parameter_name_to_parameter_mappings_and_symbolic_return_values.keys()
266:
267:     for node_providing_scope in nodes_providing_scope_set:
268:         _, _, symbolic_return_value = nodes_to_parameter_lists_parameter_name_to_parameter_mappings_and_symboli
269:
270:         apparent_return_value_set = nodes_providing_scope_to_apparent_return_value_sets[node_providing_scope]
271:         yield_value_set = nodes_providing_scope_to_yield_value_sets[node_providing_scope]
272:         send_value_set = nodes_providing_scope_to_send_value_sets[node_providing_scope]
273:
274:         # If there is no apparent return value, then add a dummy node to represent the return value of None.
275:         if not apparent_return_value_set:
276:             return_value: ast.AST = await create_new_node()
277:             await set_node_to_be_instance_of(return_value, type(None))
278:
279:             augmented_apparent_return_value_set: set[_ast.AST] = {return_value}
280:         else:
281:             augmented_apparent_return_value_set: set[_ast.AST] = apparent_return_value_set.copy()
282:
283:         # non-async functions returning generators
284:         if (
285:                 node_providing_scope in nodes_providing_scope_returning_generators
286:                 and node_providing_scope not in nodes_providing_scope_returning_coroutines
287:         ):
288:             await set_node_to_be_instance_of(symbolic_return_value, collections.abc.Generator)
```

```
289:
290:             for yield_value in yield_value_set:
291:                 await add_relation(symbolic_return_value, yield_value, NonEquivalenceRelationType.IterTargetOf)
292:
293:             for send_value in send_value_set:
294:                 await add_relation(symbolic_return_value, send_value, NonEquivalenceRelationType.SendTargetOf)
295:
296:             for apparent_return_value in augmented_apparent_return_value_set:
297:                 await add_relation(symbolic_return_value, apparent_return_value, NonEquivalenceRelationType.Yie
298:         # async functions returning generators
299:         elif (
300:                 node_providing_scope in nodes_providing_scope_returning_generators
301:                 and node_providing_scope in nodes_providing_scope_returning_coroutines
302:         ):
303:             await set_node_to_be_instance_of(symbolic_return_value, collections.abc.AsyncGenerator)
304:
305:             for yield_value in yield_value_set:
306:                 await add_relation(symbolic_return_value, yield_value, NonEquivalenceRelationType.IterTargetOf)
307:
308:             for send_value in send_value_set:
309:                 await add_relation(symbolic_return_value, send_value, NonEquivalenceRelationType.SendTargetOf)
310:         # async functions not returning generators
311:         elif (
312:                  node_providing_scope not in nodes_providing_scope_returning_generators
313:                 and node_providing_scope in nodes_providing_scope_returning_coroutines
314:         ):
315:             await set_node_to_be_instance_of(symbolic_return_value, collections.abc.Coroutine)
316:
317:             for yield_value in yield_value_set:
318:                 await add_relation(symbolic_return_value, yield_value, NonEquivalenceRelationType.IterTargetOf)
319:
320:             for send_value in send_value_set:
321:                 await add_relation(symbolic_return_value, send_value, NonEquivalenceRelationType.SendTargetOf)
322:
323:             for apparent_return_value in augmented_apparent_return_value_set:
324:                 await add_relation(symbolic_return_value, apparent_return_value, NonEquivalenceRelationType.Yie
325:         # non-async functions not returning generators
326:         else:
327:             for apparent_return_value in augmented_apparent_return_value_set:
328:                 await add_containment_relation(
329:                     superset_node=symbolic_return_value,
330:                     subset_node=apparent_return_value
331:                 )
332:
333:     # ------------------------------------------------------------------------------------------ #
334:     # The first parameter ('self') of all instance methods within a runtime class are equivalent and are instan
335:     # The first parameter ('cls') of all classmethods within a runtime class contain the class definition as a
336:     for top_level_class_definition, runtime_class in top_level_class_definitions_to_runtime_classes.items():
```

```
337:         first_parameter_of_instance_methods = set()
338:         first_parameter_of_classmethods = set()
339:
340:         for k, v in get_dict_for_runtime_class(runtime_class).items():
341:             is_staticmethod = isinstance(v, staticmethod)
342:             is_classmethod = isinstance(v, classmethod)
343:
344:             unwrapped_v = unwrap(v)
345:
346:             if (
347:                     isinstance(unwrapped_v, UnwrappedRuntimeFunction)
348:                     and unwrapped_v in unwrapped_runtime_functions_to_named_function_definitions
349:             ):
350:                 function_definition = unwrapped_runtime_functions_to_named_function_definitions[unwrapped_v]
351:
352:                 (
353:                     parameter_list,
354:                     parameter_name_to_parameter_mappings,
355:                     symbolic_return_value
356:                 ) = nodes_to_parameter_lists_parameter_name_to_parameter_mappings_and_symbolic_return_values[fu
357:
358:                 if parameter_list:
359:                     first_parameter: ast.arg = parameter_list[0]
360:
361:                     if is_classmethod:
362:                         first_parameter_of_classmethods.add(first_parameter)
363:                     if not is_staticmethod and not is_classmethod:
364:                         first_parameter_of_instance_methods.add(first_parameter)
365:
366:         # MULTI-WAY RELATION
367:         for (
368:                 first_parameter_of_instance_method_1,
369:                 first_parameter_of_instance_method_2
370:         ) in itertools.combinations(first_parameter_of_instance_methods, 2):
371:             await add_containment_relation(
372:                 superset_node=first_parameter_of_instance_method_1,
373:                 subset_node=first_parameter_of_instance_method_2
374:             )
375:             await add_containment_relation(
376:                 superset_node=first_parameter_of_instance_method_2,
377:                 subset_node=first_parameter_of_instance_method_1
378:             )
379:
380:         for first_parameter_of_instance_method in first_parameter_of_instance_methods:
381:             await set_node_to_be_instance_of(first_parameter_of_instance_method, runtime_class)
382:
383:         for first_parameter_of_classmethod in first_parameter_of_classmethods:
384:             await add_runtime_terms(first_parameter_of_classmethod, {runtime_class})
```

```
385:
386:     # ---------------------------------------------------------------------------------------------- #
387:     # Name resolution.
388:     # Resolve the names within each scope.
389:     def get_name_resolution_callback_function(module_name: str):
390:         # Keep track of what names are being defined at each scope.
391:         # None represents the global scope.
392:         nodes_providing_scope_to_local_names_to_definition_nodes: defaultdict[
393:             NodeProvidingScope | None, dict[str, ast.AST]
394:         ] = defaultdict(dict)
395:         nodes_providing_scope_to_local_names_to_definition_nodes[None].update(names_to_nodes_for_builtins)
396:         if module_name in module_names_to_names_to_dummy_ast_nodes:
397:             nodes_providing_scope_to_local_names_to_definition_nodes[None].update(
398:                 module_names_to_names_to_dummy_ast_nodes[module_name]
399:             )
400:
401:         nodes_providing_scope_to_explicit_global_names_to_definition_nodes: defaultdict[
402:             NodeProvidingScope | None, dict[str, ast.AST]
403:         ] = defaultdict(dict)
404:
405:         nodes_providing_scope_to_explicit_nonlocal_names_to_definition_nodes: defaultdict[
406:             NodeProvidingScope | None, dict[str, ast.AST]
407:         ] = defaultdict(dict)
408:
409:         async def handle_explicit_global_name_declaration(scope_stack: list[NodeProvidingScope], name: str) ->
410:             """
411:             Callback for encountered `ast.Global`'s.
412:             Adds the definition node of the name to (explicitly) global names within the current scope.
413:             """
414:             if scope_stack:
415:                 current_scope = scope_stack[-1]
416:
417:                 # Find or create definition node within the global scope.
418:                 if name in nodes_providing_scope_to_local_names_to_definition_nodes[None]:
419:                     # Directly retrieve the definition node
420:                     definition_node = nodes_providing_scope_to_local_names_to_definition_nodes[None][name]
421:                 else:
422:                     # Add a dummy node as the definition node within the global scope.
423:                     definition_node = await create_new_node()
424:                     nodes_providing_scope_to_local_names_to_definition_nodes[None][name] = definition_node
425:
426:                 # Add the definition node to (explicitly) global names within the current scope.
427:                 nodes_providing_scope_to_explicit_global_names_to_definition_nodes[current_scope][
428:                     name
429:                 ] = definition_node
430:             else:
431:                 logging.error('Cannot handle ast.Global nodes in the global scope!')
432:
```

```
433:         async def handle_explicit_nonlocal_name_declaration(scope_stack: list[NodeProvidingScope], name: str) -
434:             """
435:             Callback for encountered `ast.Nonlocal`'s.
436:             Adds the definition node of the name to (explicitly) global names within the current scope.
437:             """
438:             if scope_stack:
439:                 current_scope = scope_stack[-1]
440: 
441:                 # Find the name from parent scopes
442:                 found_definition_node = False
443: 
444:                 for scope in reversed(scope_stack[:-1]):
445:                     local_names_to_definition_nodes = nodes_providing_scope_to_local_names_to_definition_nodes[
446:                     if name in local_names_to_definition_nodes:
447:                         # Directly retrieve the definition node
448:                         definition_node = local_names_to_definition_nodes[name]
449: 
450:                         # Add the definition node to (explicitly) nonlocal names within the current scope
451:                         nodes_providing_scope_to_explicit_nonlocal_names_to_definition_nodes[current_scope][
452:                             name] = definition_node
453: 
454:                         return
455: 
456:                 if not found_definition_node:
457:                     logging.error(
458:                         'Cannot find the definition node of the nonlocal name %s given the scope stack %s!',
459:                         name, scope_stack
460:                     )
461:             else:
462:                 logging.error('Cannot handle ast.Nonlocal nodes in the global scope!')
463: 
464:         async def get_last_definition_node(
465:                 scope_stack: list[NodeProvidingScope],
466:                 name: str,
467:                 store: bool = False
468:         ) -> typing.Optional[ast.AST]:
469:             if scope_stack:
470:                 current_scope = scope_stack[-1]
471:             else:
472:                 current_scope = None
473: 
474:             last_definition_node: ast.AST | None = None
475: 
476:             # Is the name (explicitly) global within the current scope?
477:             if name in nodes_providing_scope_to_explicit_global_names_to_definition_nodes[current_scope]:
478:                 # Directly retrieve the definition node
479:                 last_definition_node = nodes_providing_scope_to_explicit_global_names_to_definition_nodes[curre
480:             # Is the name (explicitly) nonlocal within the current scope?
```

```
            elif name in nodes_providing_scope_to_explicit_nonlocal_names_to_definition_nodes[current_scope]:
                # Directly retrieve the definition node
                last_definition_node = nodes_providing_scope_to_explicit_nonlocal_names_to_definition_nodes[cur
            # Is the name local within the current scope?
            elif name in nodes_providing_scope_to_local_names_to_definition_nodes[current_scope]:
                # Directly retrieve the definition node
                last_definition_node = nodes_providing_scope_to_local_names_to_definition_nodes[current_scope][
            # The name may be (implicitly) global or nonlocal
            # In this case, the name is read
            elif not store:
                for containing_scope in itertools.chain(reversed(scope_stack[:-1]), (None,)):
                    local_names_to_definition_nodes = nodes_providing_scope_to_local_names_to_definition_nodes[
                        containing_scope]
                    if name in local_names_to_definition_nodes:
                        last_definition_node = local_names_to_definition_nodes[name]
                        break

            return last_definition_node

        async def handle_node_that_accesses_name(
                scope_stack: list[NodeProvidingScope],
                name: str,
                node: _ast.AST,
                store: bool = False
        ) -> _ast.AST:
            """
            Finds the last definition node for an accessed name under the current scope.
            If no definition node can be found,
            adds the node that accesses the name to local names within the current scope.
            """
            if scope_stack:
                current_scope = scope_stack[-1]
            else:
                current_scope = None

            last_definition_node: typing.Optional[ast.AST] = await get_last_definition_node(scope_stack, name,

            if last_definition_node is not None:
                logging.info(
                    'Found the last definition node %s for accesses name %s given the scope stack %s.',
                    last_definition_node, name, scope_stack
                )

                if store:
                    logging.info(
                        'We are storing, thus, we are redefining the name %s.',
                        name
                    )
```

```
529:
530:                     nodes_providing_scope_to_local_names_to_definition_nodes[current_scope][name] = node
531:                     return node
532:                 else:
533:                     return last_definition_node
534:             else:
535:                 # Add the node that accesses the name to local names within the current scope
536:                 if not store:
537:                     logging.error(
538:                         'Cannot find the last definition node for accessed name %s given the scope stack %s. Ad
539:                         name,
540:                         scope_stack
541:                     )
542:
543:                 nodes_providing_scope_to_local_names_to_definition_nodes[current_scope][name] = node
544:
545:                 return node
546:
547:         async def name_resolution_callback(
548:                 scope_stack: list[NodeProvidingScope],
549:                 class_stack: list[ast.ClassDef],
550:                 node: _ast.AST
551:         ):
552:             # ast.Name(id, ctx)
553:             if isinstance(node, ast.Name):
554:                 # Handle accessed name
555:                 current_or_last_definition_node = await handle_node_that_accesses_name(
556:                     scope_stack,
557:                     node.id,
558:                     node,
559:                     isinstance(node.ctx, ast.Store)
560:                 )
561:
562:                 if node != current_or_last_definition_node:
563:                     await add_two_way_containment_relation(
564:                         node,
565:                         current_or_last_definition_node
566:                     )
567:
568:                     await set_equivalent({
569:                         node,
570:                         current_or_last_definition_node
571:                     }, True)
572:             # ast.AugAssign(target, op, value)
573:             if isinstance(node, ast.AugAssign):
574:                 if isinstance(node.target, ast.Name):
575:                     last_definition_node: typing.Optional[ast.AST] = await get_last_definition_node(
576:                         scope_stack,
```

```
                        node.target.id,
                        False
                    )

                    if last_definition_node is not None:
                        await add_two_way_containment_relation(
                            node.target,
                            last_definition_node
                        )

                        await set_equivalent({
                            node.target,
                            last_definition_node,
                        }, True)
            # ast.ExceptHandler(type, name, body)
            if isinstance(node, ast.ExceptHandler):
                if node.name is not None:
                    # Handle accessed name
                    await handle_node_that_accesses_name(
                        scope_stack,
                        node.name,
                        node,
                        True
                    )
            # ast.FunctionDef(name, args, body, decorator_list, returns, type_comment)
            # ast.AsyncFunctionDef(name, args, body, decorator_list, returns, type_comment)
            if isinstance(node, (ast.FunctionDef, ast.AsyncFunctionDef)):
                # Handle accessed name.
                await handle_node_that_accesses_name(
                    scope_stack,
                    node.name,
                    node,
                    True
                )
            # ast.arguments(posonlyargs, args, vararg, kwonlyargs, kw_defaults, kwarg, defaults)
            if isinstance(node, ast.arguments):
                if scope_stack:
                    current_scope = scope_stack[-1]
                    for arg in node.posonlyargs + node.args:
                        # Handle accessed name
                        await handle_node_that_accesses_name(
                            scope_stack,
                            arg.arg,
                            arg,
                            True
                        )
                else:
                    logging.error('Cannot handle ast.arguments outside of a scope!')
```

```
625:             # ast.ClassDef(name, bases, keywords, starargs, kwargs, body, decorator_list)
626:             if isinstance(node, ast.ClassDef):
627:                 # Handle accessed name.
628:                 await handle_node_that_accesses_name(
629:                     scope_stack,
630:                     node.name,
631:                     node,
632:                     True
633:                 )
634:             if isinstance(node, ast.Global):
635:                 for name in node.names:
636:                     # Handle global name declaration
637:                     await handle_explicit_global_name_declaration(scope_stack, name)
638:             if isinstance(node, ast.Nonlocal):
639:                 for name in node.names:
640:                     # Handle nonlocal name declaration
641:                     await handle_explicit_nonlocal_name_declaration(scope_stack, name)
642: 
643:         return name_resolution_callback
644: 
645:     for module_name, module_node in module_names_to_module_nodes.items():
646:         await AsyncScopedNodeVisitor(get_name_resolution_callback_function(module_name)).visit(module_node)
647: 
648:     # ------------------------------------------------------------------------------------------ #
649:     # Visit the AST nodes bottom-up.
650:     # Handle local syntax-directed typing constraints of each AST node.
651:     async def handle_local_syntax_directed_typing_constraints_callback_function(
652:             node: ast.AST
653:     ):
654:         # ast.Constant(value)
655:         if isinstance(node, ast.Constant):
656:             # Set the current type variable to be equivalent to `type(value)`
657:             await set_node_to_be_instance_of(node, type(node.value))
658: 
659:         # ast.JoinedStr(values)
660:         if isinstance(node, ast.JoinedStr):
661:             # Set the current type variable to be equivalent to `str`
662:             await set_node_to_be_instance_of(node, str)
663: 
664:         # ast.List(elts, ctx)
665:         if isinstance(node, ast.List):
666:             # Set the current type variable to be equivalent to `list`
667:             await set_node_to_be_instance_of(node, list)
668: 
669:             for elt in node.elts:
670:                 if not isinstance(elt, ast.Starred):
671:                     # Set the type variable of `elt` as $ValueOf$ and $IterTargetOf$ the current type variable
672:                     await add_relation(node, elt, NonEquivalenceRelationType.ValueOf)
```

```
673:                 await add_relation(node, elt, NonEquivalenceRelationType.IterTargetOf)
674:
675:             # Set $KeyOf$ the current type variable to be equivalent to `int`
676:             await set_node_to_be_instance_of(
677:                 await create_related_node(node, NonEquivalenceRelationType.KeyOf),
678:                 int
679:             )
680:
681:         # ast.Tuple(elts, ctx)
682:         if isinstance(node, ast.Tuple):
683:             # Set the current type variable to be equivalent to `tuple`
684:             await set_node_to_be_instance_of(node, tuple)
685:
686:             for i, elt in enumerate(node.elts):
687:                 if not isinstance(elt, ast.Starred):
688:                     # Set the type variable of `elt` as the $i$-th $ElementOf$ the current type variable
689:                     await add_relation(node, elt, NonEquivalenceRelationType.ElementOf, i)
690:                 else:
691:                     break
692:
693:             # Set $KeyOf$ the current type variable to be equivalent to `int`
694:             await set_node_to_be_instance_of(
695:                 await create_related_node(node, NonEquivalenceRelationType.KeyOf),
696:                 int
697:             )
698:
699:         # ast.Set(elts)
700:         if isinstance(node, ast.Set):
701:             # Set the current type variable to be equivalent to `set`
702:             await set_node_to_be_instance_of(node, set)
703:
704:             for elt in node.elts:
705:                 if not isinstance(elt, ast.Starred):
706:                     # Set the type variable of `elt` as $IterTargetOf$ the current type variable
707:                     await add_relation(node, elt, NonEquivalenceRelationType.IterTargetOf)
708:
709:         # ast.Dict(keys, values)
710:         if isinstance(node, ast.Dict):
711:             # Set the current type variable to be equivalent to `dict`
712:             await set_node_to_be_instance_of(node, dict)
713:
714:             for key_, value_ in zip(node.keys, node.values):
715:                 if key_ is not None:
716:                     # Set the type variable of `key` as $KeyOf$ and $IterTargetOf$ the current type variable
717:                     await add_relation(node, key_, NonEquivalenceRelationType.KeyOf)
718:                     await add_relation(node, key_, NonEquivalenceRelationType.IterTargetOf)
719:                     # Set the type variable of `value` as $ValueOf$ the current type variable
720:                     await add_relation(node, value_, NonEquivalenceRelationType.ValueOf)
```

```
721:                 else:
722:                     # as described in https://docs.python.org/3/reference/expressions.html#dictionary-displays
723:                     # Set the type variable of `value` to be equivalent to `collections.abc.Mapping`
724:                     await set_node_to_be_instance_of(value_, collections.abc.Mapping)
725:
726:                     # Set the $KeyOf$, $ValueOf$, and $IterTargetOf$ the type variable of `value` as equivalent
727:                     await add_two_way_containment_relation(
728:                         await create_related_node(node, NonEquivalenceRelationType.KeyOf),
729:                         await create_related_node(value_, NonEquivalenceRelationType.KeyOf)
730:                     )
731:
732:                     await add_two_way_containment_relation(
733:                         await create_related_node(node, NonEquivalenceRelationType.ValueOf),
734:                         await create_related_node(value_, NonEquivalenceRelationType.ValueOf)
735:                     )
736:
737:                     # TWO-WAY RELATION
738:                     await add_two_way_containment_relation(
739:                         await create_related_node(node, NonEquivalenceRelationType.IterTargetOf),
740:                         await create_related_node(value_, NonEquivalenceRelationType.IterTargetOf)
741:                     )
742:
743:         # ast.Starred(value, ctx)
744:         if isinstance(node, ast.Starred):
745:             # Set the type variable of `value` to be equivalent to `collections.abc.Iterable`
746:             # according to https://docs.python.org/3/reference/expressions.html#grammar-token-python-grammar-st
747:             await set_node_to_be_instance_of(node.value, collections.abc.Iterable)
748:
749:         # ast.UnaryOp(op, operand)
750:         if isinstance(node, ast.UnaryOp):
751:             if not isinstance(node.op, ast.Not):
752:                 # Update the attribute counter of the type variable of `operand` with the attribute correspondi
753:                 await update_attributes(node.operand, {unaryop_to_attribute[type(node.op)]})
754:
755:                 # Set the current type variable as equivalent to the type variable of `operand`.
756:                 await add_two_way_containment_relation(
757:                     node,
758:                     node.operand
759:                 )
760:             else:
761:                 # Set the current type variable as equivalent to `bool`.
762:                 await set_node_to_be_instance_of(node, bool)
763:
764:         # ast.BinOp(left, op, right)
765:         if isinstance(node, ast.BinOp):
766:             # Update the attribute counter of the type variable of `left` with the attribute corresponding to `
767:             await update_attributes(node.left, {operator_to_attribute[type(node.op)]})
768:
```

```
769:             # Set the current type variable as equivalent to the type variable of `left`.
770:             await add_two_way_containment_relation(
771:                 node,
772:                 node.left
773:             )
774:
775:             if not isinstance(node, (ast.Mod, ast.Mult)):
776:                 await add_two_way_containment_relation(
777:                     node.left,
778:                     node.right
779:                 )
780:
781:         # ast.Compare(left, ops, comparators)
782:         if isinstance(node, ast.Compare):
783:             operands = [node.left] + node.comparators
784:             for (left, right), op in zip(
785:                     itertools.pairwise(operands),
786:                     node.ops
787:             ):
788:                 if isinstance(op, (ast.Eq, ast.NotEq, ast.Lt, ast.LtE, ast.Gt, ast.GtE)):
789:                     # Update the attribute counter of the type variable of `left` with the attribute correspond
790:                     await update_attributes(left, {cmpop_to_attribute[type(op)]})
791:
792:                     # Set the type variable of `left` as equivalent to the type variable of `right`.
793:                     await add_two_way_containment_relation(
794:                         left,
795:                         right
796:                     )
797:                 elif isinstance(op, (ast.In, ast.NotIn)):
798:                     # based on https://docs.python.org/3/reference/expressions.html#membership-test-operations
799:                     # Update the attribute counter of the type variable of `right` with the attributes `__conta
800:                     await update_attributes(right, {'__contains__', '__iter__'})
801:
802:                     # Set the type variable of `left` as $IterTargetOf$ the type variable of `right`.
803:                     await add_relation(right, left, NonEquivalenceRelationType.IterTargetOf)
804:
805:                     # Set the current type variable as equivalent to `bool`.
806:                     await set_node_to_be_instance_of(node, bool)
807:
808:         # ast.Call(func, args, keywords, starargs, kwargs)
809:         if isinstance(node, ast.Call):
810:             # Update the attribute counter of the type variable of `func` with the attribute `__call__`.
811:             await update_attributes(node.func, {'__call__'})
812:
813:             undetermined_number_of_parameters: bool = False
814:
815:             argument_node_set_list: list[set[_ast.AST]] = []
816:
```

```
817:             for i, arg in enumerate(node.args):
818:                 if not isinstance(arg, ast.Starred):
819:                     argument_node_set_list.append({arg})
820:                 else:
821:                     undetermined_number_of_parameters = True
822:                     break
823: 
824:             for keyword in node.keywords:
825:                 if keyword.arg is None:
826:                     # Set the type variable of `keyword.value` as equivalent to `collections.abc.Mapping`.
827:                     # as described in https://docs.python.org/3/reference/expressions.html#dictionary-displays
828:                     await set_node_to_be_instance_of(keyword.value, collections.abc.Mapping)
829: 
830:                     # Set the $KeyOf$ the type variable of `keyword.value` as equivalent to `str`.
831:                     await set_node_to_be_instance_of(
832:                         await create_related_node(keyword.value, NonEquivalenceRelationType.KeyOf),
833:                         str
834:                     )
835: 
836:             if node.keywords:
837:                 undetermined_number_of_parameters = True
838: 
839:             if undetermined_number_of_parameters:
840:                 # Create a dummy node to represent all parameters.
841:                 dummy_node_representing_all_parameters = await create_new_node()
842: 
843:                 await set_node_to_be_instance_of(dummy_node_representing_all_parameters, type(Ellipsis))
844: 
845:                 if argument_node_set_list:
846:                     argument_node_set_list[0].add(dummy_node_representing_all_parameters)
847:                 else:
848:                     argument_node_set_list.append({dummy_node_representing_all_parameters})
849: 
850:             # Set the type variable of `arg` as the $i$-th $ParameterOf$ the type variable of `func`.
851:             # Set the current type variable as the $ReturnValueOf$ the type variable of `func`.
852:             await add_argument_of_returned_value_of_relations(
853:                 node.func,
854:                 argument_node_set_list,
855:                 {node}
856:             )
857: 
858:         # ast.IfExp(test, body, orelse)
859:         if isinstance(node, ast.IfExp):
860:             await add_containment_relation(
861:                 superset_node=node,
862:                 subset_node=node.body
863:             )
864: 
```

```
865:             await add_containment_relation(
866:                 superset_node=node,
867:                 subset_node=node.orelse
868:             )
869: 
870:         # ast.Attribute(value, attr, ctx)
871:         if isinstance(node, ast.Attribute):
872:             # Update the attribute counter of the type variable of `value` with `attr`.
873:             await update_attributes(node.value, {node.attr})
874: 
875:             # Set the current type variable as the $attr$-$AttrOf$ the type variable of `value`.
876:             await add_relation(node.value, node, NonEquivalenceRelationType.AttrOf, node.attr)
877: 
878:         # ast.NamedExpr(target, value)
879:         if isinstance(node, ast.NamedExpr):
880:             # Set the current type variable as equivalent to the type variable of `target` and `value`.
881:             await add_two_way_containment_relation(
882:                 node.target,
883:                 node.value
884:             )
885: 
886:             await add_two_way_containment_relation(
887:                 node,
888:                 node.target
889:             )
890: 
891:             await set_equivalent({
892:                 node,
893:                 node.target,
894:                 node.value
895:             }, True)
896: 
897:         # ast.Subscript(value, slice, ctx)
898:         if isinstance(node, ast.Subscript):
899:             if isinstance(node.slice, (ast.Tuple, ast.Slice)):
900:                 # Set the current type variable as equivalent to the type variable of `value`.
901:                 await add_two_way_containment_relation(
902:                     node,
903:                     node.value
904:                 )
905:             else:
906:                 # Set the current type variable as $ValueOf$ the type variable of `value`.
907:                 await add_relation(node.value, node, NonEquivalenceRelationType.ValueOf)
908: 
909:                 # Set the type variable of `slice` as $KeyOf$ the type variable of `value`.
910:                 await add_relation(node.value, node.slice, NonEquivalenceRelationType.KeyOf)
911: 
912:             if isinstance(node.ctx, ast.Load):
```

```
913:                 # Update the attribute counter of the type variable of `value` with the attribute `__getitem__`
914:                 await update_attributes(node.value, {'__getitem__'})
915:
916:             if isinstance(node.ctx, ast.Store):
917:                 # Update the attribute counter of the type variable of `value` with the attribute `__setitem__`
918:                 await update_attributes(node.value, {'__setitem__'})
919:
920:         # ast.Slice(lower, upper, step)
921:         if isinstance(node, ast.Slice):
922:             # Set the current type variable as equivalent to `slice`.
923:             await set_node_to_be_instance_of(node, slice)
924:
925:             for value in (node.lower, node.upper, node.step):
926:                 if value is not None:
927:                     # Set the type variable of `value` as equivalent to `int`.
928:                     await set_node_to_be_instance_of(value, int)
929:
930:         # ast.ListComp(elt, generators)
931:         if isinstance(node, ast.ListComp):
932:             # Set the current type variable as equivalent to `list`.
933:             await set_node_to_be_instance_of(node, list)
934:
935:             # Set the type variable of `elt` as $ValueOf$ and $IterTargetOf$ the current type variable.
936:             await add_relation(node, node.elt, NonEquivalenceRelationType.ValueOf)
937:             await add_relation(node, node.elt, NonEquivalenceRelationType.IterTargetOf)
938:
939:             # Set $KeyOf$ the current type variable as equivalent to `int`.
940:             await set_node_to_be_instance_of(
941:                 await create_related_node(node, NonEquivalenceRelationType.KeyOf),
942:                 int
943:             )
944:
945:         # ast.SetComp(elt, generators)
946:         if isinstance(node, ast.SetComp):
947:             # Set the current type variable as equivalent to `set`.
948:             await set_node_to_be_instance_of(node, set)
949:
950:             # Set the type variable of `elt` as $IterTargetOf$ the current type variable.
951:             await add_relation(node, node.elt, NonEquivalenceRelationType.IterTargetOf)
952:
953:         # ast.GeneratorExp(elt, generators)
954:         if isinstance(node, ast.GeneratorExp):
955:             # Set the current type variable as equivalent to `collections.abc.Generator`.
956:             await set_node_to_be_instance_of(node, collections.abc.Generator)
957:
958:             # Set the type variable of `elt` as $IterTargetOf$ the current type variable.
959:             await add_relation(node, node.elt, NonEquivalenceRelationType.IterTargetOf)
960:
```

```
        # ast.DictComp(key, value, generators)
        if isinstance(node, ast.DictComp):
            # Set the current type variable as equivalent to `dict`.
            await set_node_to_be_instance_of(node, dict)

            # Set the type variable of `key` as $KeyOf$ and $IterTargetOf$ the current type variable.
            await add_relation(node, node.key, NonEquivalenceRelationType.KeyOf)
            await add_relation(node, node.key, NonEquivalenceRelationType.IterTargetOf)

            # Set the type variable of `value` as $ValueOf$ the current type variable.
            await add_relation(node, node.value, NonEquivalenceRelationType.ValueOf)

        # ast.comprehension(target, iter, ifs, is_async)
        if isinstance(node, ast.comprehension):
            if node.is_async:
                # Update the attribute counter of the type variable of `iter` with the attribute `__aiter__`.
                await update_attributes(node.iter, {'__aiter__'})
            else:
                # Update the attribute counter of the type variable of `iter` with the attribute `__iter__`.
                await update_attributes(node.iter, {'__iter__'})

            # Set the type variable of `target` as $IterTargetOf$ the type variable of `iter`.
            await add_relation(node.iter, node.target, NonEquivalenceRelationType.IterTargetOf)

        # ast.Assign(targets, value, type_comment)
        if isinstance(node, ast.Assign):
            async def _r(
                    targets: list[ast.expr],
                    value: ast.expr
            ):
                if targets:
                    targets_up_front, last_target = targets[:-1], targets[-1]

                    await add_two_way_containment_relation(
                        last_target,
                        value
                    )

                    await _r(
                        targets_up_front,
                        last_target
                    )

            await _r(
                node.targets,
                node.value
            )
```

```
1009:             await set_equivalent({
1010:                 node.value,
1011:                 *node.targets
1012:             }, True)
1013:
1014:         # ast.AnnAssign(target, annotation, value, simple)
1015:         if isinstance(node, ast.AnnAssign):
1016:             if node.value is not None:
1017:                 await add_two_way_containment_relation(
1018:                     node.target,
1019:                     node.value
1020:                 )
1021:
1022:                 await set_equivalent({
1023:                     node.target,
1024:                     node.value
1025:                 }, True)
1026:
1027:         # ast.AugAssign(target, op, value)
1028:         if isinstance(node, ast.AugAssign):
1029:             # Update the attribute counter of the type variable of `target` with the attribute corresponding to
1030:             await update_attributes(node.target, {operator_to_attribute[type(node.op)]})
1031:
1032:             await add_two_way_containment_relation(
1033:                 node.target,
1034:                 node.value
1035:             )
1036:
1037:         # ast.For(target, iter, body, orelse, type_comment)
1038:         if isinstance(node, ast.For):
1039:             # Update the attribute counter of the type variable of `iter` with the attribute `__iter__`.
1040:             await update_attributes(node.iter, {'__iter__'})
1041:
1042:             # Set the type variable of `target` as $IterTargetOf$ the type variable of `iter`.
1043:             await add_relation(node.iter, node.target, NonEquivalenceRelationType.IterTargetOf)
1044:
1045:         # ast.AsyncFor(target, iter, body, orelse, type_comment)
1046:         if isinstance(node, ast.AsyncFor):
1047:             # Update the attribute counter of the type variable of `iter` with the attribute `__aiter__`.
1048:             await update_attributes(node.iter, {'__aiter__'})
1049:
1050:             # Set the type variable of `target` as $IterTargetOf$ the type variable of `iter`.
1051:             await add_relation(node.iter, node.target, NonEquivalenceRelationType.IterTargetOf)
1052:
1053:         # ast.With(items, body, type_comment)
1054:         if isinstance(node, ast.With):
1055:             for withitem in node.items:
1056:                 # Update the attribute counter of the type variable of `withitem.context_expr` with the attribu
```

```
1057:                 await update_attributes(withitem.context_expr, {'__enter__', '__exit__'})
1058:
1059:                 if withitem.optional_vars is not None:
1060:                     # 'getattr_node = ast.Attribute(value=withitem.context_expr, attr='__enter__', ctx=ast.Load
1061:                     getattr_node = await create_new_node()
1062:
1063:                     await add_relation(
1064:                         withitem.context_expr,
1065:                         getattr_node,
1066:                         NonEquivalenceRelationType.AttrOf,
1067:                         '__enter__'
1068:                     )
1069:
1070:                     # Set the type variable of 'withitem.optional_vars' as the $ReturnValueOf$ the type variabl
1071:                     await add_relation(
1072:                         getattr_node,
1073:                         withitem.optional_vars,
1074:                         NonEquivalenceRelationType.ReturnedValueOf
1075:                     )
1076:
1077:         # ast.AsyncWith(items, body, type_comment)
1078:         if isinstance(node, ast.AsyncWith):
1079:             for withitem in node.items:
1080:                 # Update the attribute counter of the type variable of 'withitem.context_expr' with the attribu
1081:                 await update_attributes(withitem.context_expr, {'__aenter__', '__aexit__'})
1082:
1083:                 if withitem.optional_vars is not None:
1084:                     # 'getattr_node = ast.Attribute(value=withitem.context_expr, attr='__aenter__', ctx=ast.Loa
1085:                     getattr_node = await create_new_node()
1086:
1087:                     await add_relation(
1088:                         withitem.context_expr,
1089:                         getattr_node,
1090:                         NonEquivalenceRelationType.AttrOf,
1091:                         '__aenter__'
1092:                     )
1093:
1094:                     # Set the type variable of 'withitem.optional_vars' as the $YieldFromAwaitResultOf$ the $Re
1095:                     await add_relation(
1096:                         await create_related_node(getattr_node, NonEquivalenceRelationType.ReturnedValueOf),
1097:                         withitem.optional_vars,
1098:                         NonEquivalenceRelationType.YieldFromAwaitResultOf
1099:                     )
1100:
1101:         # ast.YieldFrom(value)
1102:         if isinstance(node, ast.YieldFrom):
1103:             # Set the current type variable as the $YieldFromAwaitResultOf$ the type variable of 'value'.
1104:             await add_relation(node.value, node, NonEquivalenceRelationType.YieldFromAwaitResultOf)
```

```
1105:
1106:             # Update the attribute counter of the type variable of `value` with the attribute `__iter__`.
1107:             await update_attributes(node.value, {'__iter__'})
1108:         # ast.Await(value)
1109:         if isinstance(node, ast.Await):
1110:             # Update the attribute counter of the type variable of `value` with the attribute `__await__`.
1111:             await update_attributes(node.value, {'__await__'})
1112:
1113:             # Set the current type variable as the $YieldFromAwaitResultOf$ of the type variable of `value`
1114:             await add_relation(node.value, node, NonEquivalenceRelationType.YieldFromAwaitResultOf)
1115:
1116:     for module_name, module_node in module_names_to_module_nodes.items():
1117:         await AsyncEvaluationOrderBottomUpNodeVisitor(handle_local_syntax_directed_typing_constraints_callback_
```