CISPA HELMHOLTZ CENTER FOR INFORMATION SECURITY

Automated Debugging

WS 2022/2023

Prof. Dr. Andreas Zeller
Paul Zhu
Marius Smytzek

Exercise 6 (10 Points)

Due: 18. January 2023

Submit your solutions as a Zip file on your status page in the CMS.

We will provide you a structure to submit your solutions where each task has a dedicated file. You can add new files and scripts if you want, but you must not delete any provided ones. You can verify whether your submission is valid by calling python3.

```
python3 verify.py
```

The output provides an overview if a required file, variable, or function is missing and if a function pattern was altered. If you do not follow this structure or change it, we cannot evaluate your submission. A non evaluable exercise will result in 0 points, so make sure to verify your work before submitting it. Note that the script does not reveal if your solutions are correct.

Exercise 6-1: Hello, Are You Still Alive? (2 Points)

Do you remember Exercise 3-1? Please reread this exercise.

In **exercise_1.py**, you can find the implementation of the heartbeat protocol again. Your goal for this exercise is to add pre- and postconditions such that the program aborts the execution when the input triggers the Heartbleed vulnerability. You have to use the `precondition` and `postcondition` decorator from Mining Function Specifications to introduce the conditions.

For this exercise, you cannot assume that `store_data()` and `get_data()` are correct (indeed, for evaluating your submission, we will modify these functions). However, for this exercise, you must add pre- and postcondition to these functions to catch all bugs.

Hint: You need to add one precondition and three postconditions.

In [1]:
```
data = 'password:hjasdiebk456jhaccount:smytzek'

def store_data(payload: str):
    global data
    data = payload + data

def get_data(length: int) -> str:
    return data[:min(length, len(data) + 1)]

def heartbeat(length: int, payload: str) -> str:
    store_data(payload)
    return get_data(length)
```

Exercise 6-2: Learning Is the New Black (2 Points)

In this exercise, you must demonstrate your knowledge of how to apply the learned inference

techniques.

a. Types (0.5 Points)

First, you must apply the `TypeAnnotator` to infer the types of the `mystery()` function. Use the `test_mystery()` function to execute the tests. Remember that we may modify this function during the evaluation to verify your result.

In [25]:
```
def mystery(x, y):
    if len(y) > 0:
        return x * y
    else:
        raise ValueError('len(y) <= 0')
```

In [26]:
```
def test_mystery():
    mystery(1, 'test')
    mystery(-1, 'test')
```

Implement the `run() -> TypeAnnotator` function in **exercise_2a.py** that creates a `TypeAnnotator`, runs the `test_mystery`, and returns the created `TypeAnnotator`.

You can execute this script, and the output should look like this:

```
def mystery(x: int, y: str) -> str:
    if len(y) > 0:
        return x * y
    else:
        raise ValueError('len(y) <= 0')
```

b. Invariants (1.5 Points)

Now, you must apply the `InvariantAnnotator` to infer the pre- and postconditions of the `mystery()` function. Use the `test_mystery()` function to execute the tests. Again we may modify this function during the evaluation to verify your result.

Implement the `run() -> InvariantAnnotator` function in **exercise_2b.py**. You should add invariant patterns that allow the learning of whether an argument's length (`len`) is larger than `0` and an argument is of type `str`. Creates an `InvariantAnnotator` that considers all patterns of the DebuggingBook and the two you implemented. Run the `test_mystery` and returns the created `InvariantAnnotator`.

You can execute the script, and the output should look like this:

```
@precondition(lambda x, y: isinstance(x, int))
@precondition(lambda x, y: isinstance(y, str))
@precondition(lambda x, y: len(y) > 0)
@precondition(lambda x, y: x)
@precondition(lambda x, y: y)
@postcondition(lambda return_value, x, y: isinstance(return_value, str))
@postcondition(lambda return_value, x, y: return_value <= y)
@postcondition(lambda return_value, x, y: return_value == x * y)
@postcondition(lambda return_value, x, y: return_value == y * x)
@postcondition(lambda return_value, x, y: return_value in y)
@postcondition(lambda return_value, x, y: y >= return_value)
@postcondition(lambda return_value, x, y: y.endswith(return_value))
@postcondition(lambda return_value, x, y: y.startswith(return_value))
def mystery(x, y):
    if len(y) > 0:
        return x * y
    else:
        raise ValueError('len(y) <= 0')
```

Exercise 6-3: You Are Not My Type! (6 Points)

For this exercise, you will implement a simple type checker that verifies that types match in the order of execution (forward). The type checker will work on a subset of the Python language and leverages `ast` to accomplish the checking as a `ast.NodeVisitor`. You will implement your solution in **exercise_3.py** within the `ForwardTypeChecker` class.

Before explaining this exercise, here are some hints you should keep in mind while reading it:

- You do not need to add new methods to the `ForwardTypeChecker`.
- You only have to add code directly after each of the `# TODO: ...` comments.
- You only have to consider three types, which we explain below.

This exercise relies on our own typing system, allowing us to compare types easily. Hence, we introduce the following `Type` class:

In [32]:
```
class Type:
    def __eq__(self, other):
        return isinstance(other, Anything) or type(self) == type(other)

    def __repr__(self) -> str:
        return self.__class__.__name__

    def __str__(self) -> str:
        return self.__repr__()
```

As you can see, this already uses one of our types, `Anything`:

In [33]:
```
class Anything(Type):
    def __eq__(self, other):
        return isinstance(other, Type)
```

`Anything` represents a type that could be any type. The other two types are `Int` and `Str`:

In [34]:
```
class Int(Type):
    pass

class Str(Type):
    pass
```

For storing types, we introduce the `TypeScope` class that holds all types for all names in a dictionary. You can create a new variable in the scope by calling the `put(str, Type)` method (for arguments and annotated assigns). Besides, you can update an existing variable with `update(str, Type)` (for normal assigns), which also creates the variable if it does not already exist. You can use the `get(str)` method to retrieve the type for a name or None if the name does not occur in the scope.

In [35]:
```
class TypeScope:

    def __init__(self, parent=None):
        self.parent = parent
        self.types: Dict[str, Type] = dict()

    def enter(self) -> Any:
        return TypeScope(self)

    def exit(self) -> Any:
        return self.parent if self.parent else self

    def put(self, name: str, type_: Type) -> None:
        self.types[name] = type_


    def get(self, name: str) -> Optional[Type]:
```

```
        if name in self.types:
            return self.types[name]
        elif self.parent:
            return self.parent.get(name)
        else:
            return None

    def update(self, name: str, type_: Type):
        if name in self.types:
            self.types[name] = type_
            return True
        elif self.parent:
            result = self.parent.update(name, type_)
            if not result:
                self.put(name, type_)
        return False
```

Besides this variable scope, we introduce the `FunctionScope` class that stores the argument types and the function's return type. You do not have to handle this scope. We already provide the code that deals with the `FunctionScope` in the `ForwardTypeChecker`.

In [36]:
```
class FunctionScope:

    def __init__(self):
        self.types: Dict[str, Tuple[List[Type], Type]] = dict()

    def put(self, name: str, types: Tuple[List[Type], Type]) -> None:
        self.types[name] = types


    def get(self, name: str) -> Optional[Tuple[List[Type], Type]]:
        if name in self.types:
            return self.types[name]
        else:
            return None
```

Now you need to implement the TODOs in the `ForwardTypeChecker` class:

In [37]:
```
class ForwardTypeChecker(ast.NodeVisitor):

    def __init__(self):
        self.scope = TypeScope()
        self.functions = FunctionScope()
        self.current_return = None

    def generic_visit(self, node: ast.AST) -> Optional[Type]:
        raise SyntaxError(f'Unsupported node {node.__class__.__name__}')

    def visit_Module(self, node: ast.Module) -> Optional[Type]:
        for n in node.body:
            self.visit(n)
        return None

    def visit_FunctionDef(self, node: ast.FunctionDef) -> Optional[Type]:
        assert self.current_return is None, \
            f'Nested function {node.name}'
        self.scope = self.scope.enter()
        types = list()

        # TODO: iterate over args, add the types (arg.annotation) in
        # order of appearance to the types variable, and put the types
        # in the scope.

        returns = Anything()

        # TODO: find the correct value for returns that matches the
        # function return value

        self.functions.put(node.name, (types, returns))
        self.current_return = returns
```

```
        # TODO: iterate over node.body to perform a type check of the
        # body

        self.current_return = None
        self.scope = self.scope.exit()
        return None

    def visit_Assign(self, node: ast.Assign) -> Optional[Type]:
        assert len(node.targets) == 1, \
            'Targets is longer than 1'
        assert isinstance(node.targets[0], ast.Name), \
            'Target is not a Name'
        target = node.targets[0]

        # TODO: find the type of the assigned expression and update the
        # target's (target.id) type in self.scope

        return None

    def visit_AnnAssign(self, node: ast.AnnAssign) -> Optional[Type]:
        assert isinstance(node.target, ast.Name), \
            'Target is not a Name'
        assert isinstance(node.annotation, ast.Name), \
            'Annotation is not a Name'
        assert node.value is not None, \
            'Value is None'

        # TODO: find the type of the assigned expression and the type of
        # the target (hint: node.annotation) and check if they match

        return None

    def visit_Expr(self, node: ast.Expr) -> Optional[Type]:
        # TODO: check the type of the expression

        return None

    def visit_Return(self, node: ast.Return) -> Optional[Type]:
        # TODO: get the type of the return value and compare it to the
        # current_return type

        return None

    def visit_Name(self, node: ast.Name) -> Optional[Type]:
        # TODO: get the type of the node (node.id) and raise a TypeError
        # if it does not exist
        # Return the type of the expression

        pass

    def visit_BinOp(self, node: ast.BinOp) -> Optional[Type]:
        assert (isinstance(node.op, ast.Add) or
                isinstance(node.op, ast.Mult)), \
            f'Unsupported op {node.op.__class__.__name__}'
        # TODO: get the types of left and right, check that they match
        # with the operator Add (+) or Mult (*)
        # Return the type of the expression

        pass

    def visit_Call(self, node: ast.Call) -> Optional[Type]:
        assert isinstance(node.func, ast.Name), \
            'Func is not a Name'
        expected_types = self.functions.get(node.func.id)
        assert expected_types is not None, \
            'Func not defined'
        expected_types, return_type = expected_types
        assert len(expected_types) == len(node.args), \
            'Number of args do not match'

        # TODO: check that the expected types match with the given
        # arguments
```

```
        return return_type

    def visit_Constant(self, node: ast.Constant) -> Optional[Type]:
        # TODO: check the type of the value and return our Type
        # format that matches
        pass
```

The `ForwardTypeChecker` performs the type checking when calling `visit(ast.Ast)` on the AST of the program to check. The check raises a `TypeError` if types do not match or the scopes do not have the type for a name.

We will go to each function and explain what you need to do.

When visiting an `ast.FunctionDef`, you need to iterate over its arguments to extract the types in the following fashion: If the arg's annotation is `'int'`, you should use our `Int` type. If the arg's annotation is `'str'`, you should use our `Str` type. If the arg's annotation is `None` or something else, you should use the `Anything` type. Add the arguments in order of appearance to the `types` variable to create the function's argument signature. You must add the arguments with their type to `self.scope` by using the `put(str, Type)` method. Next, you must check the function's return type in the same fashion as for arguments and store it in the `returns` variable.

When visiting an `ast.Assign`, you need to evaluate and get the type of the assigning expression by visiting it. Update the `self.scope` type for the `target.id`.

When visiting an `ast.AnnAssign`, you need to get the target's type with the help of `node.annotation`, evaluate and get the type of the assigning expression, and check if they match (`==`). Then must put the type for `target.id` in `self.scope`.

When visiting an `ast.Expr`, you need to evaluate the type of the wrapped expression by visiting it.

When visiting an `ast.Return`, you need to evaluate and get the type of the returned expression and verify it against `self.current_return`.

When visiting an `ast.Name`, you must get its type with `self.scope`. If the type is not in the scope, i.e., it is `None`, raise a `TypeError`.

When visiting an `ast.BinOp`, you only need to consider the `ast.Add` (`+`) and `ast.Mult` (`*`) operators. You require the types of both operands by visiting them. Then you need to evaluate them based on the following tables and return the resulting type:

+	type_1 = Int	type_1 = Str	type_1 = Anything
type_2 = Int	Int	TypeError	Int
type_2 = Str	TypeError	Str	Str
type_2 = Anything	Int	Str	Anything

*	type_1 = Int	type_1 = Str	type_1 = Anything
type_2 = Int	Int	Str	Int
type_2 = Str	Str	TypeError	Str
type_2 = Anything	Int	Str	Anything

When visiting an `ast.Call`, you need to get the types of all arguments and compare them to the `expected_types` for the arguments. Then you need to return the `return_type`.

When visiting an `ast.Constant`, you need to check what type the constant's value has (Python's `int`, `str`, or something else) and return the corresponding type of our typing system.

You should only return a Type for the expressions `ast.Name`, `ast.BinOp`, `ast.Call`, and `ast.Constant`. The other statements do not need to return anything (returning `None`).

We provide you with some tests that verify your implementation. You can execute them by running the following:

```
python3.10 exercise_3.py
```

Hint: We will test all functionality individually and combined, and we will also have secret tests for this exercise (as it could be with all other exercises). You can develop your own test ideas and share them with other students as long as you do not share parts of your solution. The best is to leverage the `test(str, bool)` function that takes the source of a program, and a flag that is set to `True` is the test should trigger a `TypeError`.