GeneBench Transformation Operators

GeneBench implements 22 transformation operators, which uses them in a multi-objective evolutionary algorithm to transform any arbitrary programming benchmark to a more complex, yet readable version of it. This document explains the details about each transformation operator, accompanied with an example. Note that operators use syntactic identifier names, and these names will be replaced by a natural alternatives later.

1 Code Structures

AddNestedFor. `AddNestedFor` introduces a nested `for` to an existing `for` loop. Additionally, it incorporates two assignments that generate randomly consecutive positive integers, which are used to define the `range` function for the iterable. The following code snippet shows an example of this operator.

```
+   checker1 = 465
+   checker2= 464
+   for index in range(checker1 // checker2):
        for H in Hs:
            maxH = H
```

AddNestedIf. `AddNestedIf` adds an additional `if` statement outside an existing `if`, creating a nested `if` structure. It also includes assignments with two random positive integers to form a predicate that consistently evaluates to `True`. The following code snippet is an example demonstrating the use of this operator.

```
+    checker1 = 352
+    checker2 = 647
+    if checker1 & checker2:
         if X > t:
             answer = X - t
```

AddNestedWhile. The `AddNestedWhile` operator introduces a nested `while` loop to an existing `while`. Similar to other operators that create nested structures, it generates two random consecutive positive integers to establish the condition for the loop. The code snippet below is an example demonstrating this operator.

```
+   checker1 = 665
+   checker2 = 664
+   while checker1 % checker2 == 1:
+       checker1 += 1
        while cnt <= 9:
            # omited code
            cnt += 1
```


Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

AddThread. Python threading enables concurrent execution of different parts of a program and is commonly employed in real-world applications. The operator `AddThread` extracts function calls and initiates a new thread for the identified function to simulate the parallel processing in real-world programs. The following code snippet demonstrates the use of this operator.

```
+   import threading
+   import queue
    def get_jwt_user(request):
        # omitted code...
        return token
-   token = get_jwt_user(request)
+   def get_jwt_user_thread(queue):
+           result = get_jwt_user(request)
+           queue.put(result)
+   thread_get_jwt_user0 = threading.Thread(
+   target=get_jwt_user_thread,
+                  args=(queue_get_jwt_user0,))
+   thread_get_jwt_user0.start()
+   thread_get_jwt_user0.join()
+   token = queue_get_jwt_user0.get()
```

AddTryExcept. Code construct `try-except` is commonly used in real-world Python programs to handle exceptions. The operator `AddTryExcept` identifies functions that lack a `try-except` handler and automatically adds one, simulating error-handling capabilities. The code snippet below shows an example of this operator.

```
    def cal(k, a, b):
+       try:
            if k % a == 0:
                print(a - b)
+       except:
+           pass
```

CreateFunction. Intra-class dependencies are prevalent in Python programs. The operator `CreateFunctions` creates intra-dependency within a Python program by extracting assignments with operators on the right-hand side (RHS), identifying the arguments, and encapsulating them into new functions. Unlike [3], which only supports cases where the left-hand side (LHS) is a single variable and the RHS consists solely of primitive variables (e.g., `a = b + c`), our approach extends support to more complex cases including where the RHS includes constants or list elements. Specifically, we identify non-variable components, initialize them as variables first, and then generalize the function accordingly. Below is an example only supported by GeneBench.

```
+   def AddElements(h, H, w, W, var0, var1):
+       return H * W - (h * W + w * H - h * w) + var0 + var1

-   ans = H * W - (h * W + w * H - h * w) + 1 + List[0]
+   var0 = 1
+   var1 = List[0]
+   ans = AddElements(h, H, w, W, var0, var1)
```

Table 1: GeneBench transformation operators.

Type	ID	Transformation Operator	Description	Analysis Sensitivity	In Previous Work
Code Structures	S_1	`AddNestedFor`	Add another nested `for` to an existing `for` loop	Flow	No
	S_2	`AddNestedIf`	Add another nested `if` to an existing `if` statement	Flow	No
	S_3	`AddNestedWhile`	Add another nested `while` to an existing `while` loop	Flow	No
	S_4	`AddThread`	Introduce a `thread` integrated with `queue`	Flow & Context	No
	S_5	`AddTryExcept`	Add a `try-except` handler inside existing functions	Flow	No
	S_6	`CreateFunction`	Transform existing statements into new functions	Flow & Context	[3]
	S_7	`CreateModuleDependencies`	Move functions into another Python file	Flow & Context	No
	S_8	`IntroduceDecorator`	Introduce a decorator	Flow & Context	No
	S_9	`ReplaceNumpy`	Replace applicable built-in calculations with `Numpy`	Flow & Context	No
	S_{10}	`TransformAugAssignment`	Transform augment assignment to a normal assignment	Flow	[3]
	S_{11}	`TransformLoopToRecursion`	Transform existing `for` loop into recursive functions	Flow & Context	No
	S_{12}	`TransformPrimToCompound`	Transform variables of primitive types into compound types, e.g., lists	Flow	No
API Calls	A_1	`AddBase64`	Introduce API calls from `base64` library	Context	No
	A_2	`AddCrypto`	Introduce API calls from `cryptography` library	Context	No
	A_3	`AddDatetime`	Introduce API calls from `datetime` library	Context	No
	A_4	`AddDateutil`	Introduce API calls from `dateutil` library	Context	No
	A_5	`AddHttp`	Introduce `http` connections	Context	No
	A_6	`AddScipy`	Introduce API calls from `scipy` library	Context	No
	A_7	`AddSklearn`	Introduce API calls from `sklearn` library	Context	No
	A_8	`AddTime`	Introduce API calls from `time` library	Context	No
Renaming	N_1	`RenameVariable`	Rename existing variables	Flow	[1–5]
	N_2	`RenameFunction`	Rename existing functions	Flow & Context	[2–4]

CreateModuleDependencies. Inter-class dependencies refer to function calls across different Python files. The transformation operator `CreateModuleDependencies` introduces inter-class dependencies by moving functions into another module file and adding a new call. The code snippet below shows an example of this operator.

```
+   from lcm_list_Class import lcm_list
-   # moved to class lcm_list_Class
-   def lcm_list(numbers):
-       return reduce(lcm, numbers, 1)
    my_lcm = lcm_list(A)
```

IntroduceDecorator. In Python, a decorator is a function that extends the behavior of another function without explicitly modifying its body and is commonly employed in practical programming. The operator `IntroduceDecorator` creates decorators for existing functions in Python code. Below is an example of this operator.

```
+   def my_decorator(func):
+       def wrapper(*args, **kwargs):
+           res = func(*args, **kwargs)
+           return res
+       return wrapper
+   @my_decorator
    def calculate():
        A.sort()
```

ReplaceNumpy. Operator `ReplaceNumpy` replaces applicable functions with `numpy` library. The following code snippet (i) is an example of this operator, where the original built-in function `min()` is replaced with `numpy.min()`, and (A[i], B[i]) is converted to array `numpy.array([[i], B[i]])`. The following is an example of this operator.

```
+   import numpy as np
-   d = min(A[i], B[i])
+   d = np.min(np.array([A[i], B[I]]))
```

TransformAugAssignment. Operator `TransformAugAssignment` converts an existing `Augment` assignment into a normal assignment. The code snippet below is an example of this operator.

```
-   a += 1
+   a = a + 1
```

TransformLoopToRecursion. Operator `TransformLoopToRecursion` converts an existing `for` loop that uses `range()` function into a recursive function. The following code snippet is an example of this operator.

```
+    def recursive_function(i):
+        if i >= 5:
+            return None
+        if str(a[i])[-1] == '0': \
+            dic.update({i: 0})
+        else:
+            dic.update({i: \
+           int(str(a[i])[-1]) - 10})
+        recursive_function(i + 1)

-   for i in range(5):
-   if str(a[i])[-1] == "0":
-       dic.update({i: 0})
-   else:
-       dic.update({i: int(str(a[i])
- [-1]) - 10})

+   recursive_function(0)
```

TransformPrimToCompound. Operator `TransformPrimToCompound` converts an existing primitive type variable into a compound one to increase type complexity in the program. The following code snippet shows an example.

```
-   a = 0
+   a = [0][0]
```

2 Third-Party API Calls

This operator adds a new API call and corresponding import statements to it if they do not exist. GeneBench currently supports adding the following libraries: `base64`, `cryptography.fernet`, `datetime`, `dateutil.parser`, `http.client`, `scipy.stats`, `sklearn.utils`, and `time`. Below are examples of these operators. All required arguments are randomly generated using a seed based on the current date and time.

```
import base64
base64.b64encode(b'Random String')
```

AddBase64

```
from cryptography.fernet import Fernet
Fernet.generate_key()
```

AddCrypto

```
import datetime
datetime.datetime.now()
```

AddDatetime

```
from dateutil.parser import parse
parse({Random_Date})
```

AddDateutil

```
from http.client import HTTPConnection
HTTPConnection('google.com', port=80)
```

AddHttp

```
from scipy.stats import ttest_ind
ttest_ind([86, 97, 97], [23, 83, 86])
```

AddScipy

```
from sklearn.utils import shuffle
shuffle([14, 54, 55])
```

AddSklearn

```
import time
time.sleep(0.02)
```

AddTime

3 Renaming

Renaming operators first identify the control flow graph and AST nodes of a program and its inter-dependent files, extract variables and functions along with their respective locations, and then perform renaming.

4 Semantically Altering Operators

We developed *five* operators to modify program semantics and applied high-order mutation to originally correct programs, injecting multiple bugs per program and causing test failures. The following table provides their descriptions.

Transformation Operator	Description
`ChangeArithOperator`	Change arithmetic operators, e.g., from `a + b` to `a - b`.
`ChangeCompareOperator`	Change comparison operators, e.g., from `a > b` to `a < b`.
`ChangeLogicalOperator`	Change logical operators, e.g., from `a and b` to `a or b`.
`ChangeReturnStat`	Reverse boolean value of a `return` statement.
`ChangeVarType`	Change types of variables, e.g., from `a = 1` to `a = '1'`.

References

[1] Saikat Chakraborty, Toufique Ahmed, Yangruibo Ding, Premkumar T Devanbu, and Baishakhi Ray. 2022. Natgen: generative pre-training by "naturalizing" source code. In *Proceedings of the 30th ACM joint european software engineering conference and symposium on the foundations of software engineering*. 18–30.

[2] Shing-Chi Cheung Jialun Cao, Wuqi Zhang. 2024. Concerned with Data Contamination? Assessing Countermeasures in Code Language Model. *arXiv preprint arXiv:2403.16898* (2024).

[3] Yaoxian Li, Shiyi Qi, Cuiyun Gao, Yun Peng, David Lo, Zenglin Xu, and Michael R Lyu. 2022. A closer look into transformer-based code intelligence through code transformation: Challenges and opportunities. *arXiv preprint arXiv:2207.04285* (2022).

[4] Maryam Vahdat Pour, Zhuo Li, Lei Ma, and Hadi Hemmati. 2021. A search-based testing framework for deep neural networks of source code embedding. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 36–46.

[5] Md Rafiqul Islam Rabin, Nghi DQ Bui, Ke Wang, Yijun Yu, Lingxiao Jiang, and Mohammad Amin Alipour. 2021. On the generalizability of neural program models with respect to semantic-preserving program transformations. *Information and Software Technology* 135 (2021), 106552.