Invariants

Being able to reason about the correctness of your programs is an important skill for any programmer. This type of reasoning allows for more systematic design of programs and usually produces cleaner code with fewer errors. The building blocks for this reasoning are logical *conditions* that consist of Boolean expressions involving values in the program. For these notes, we model an invariant as a Boolean expression `invariant(...)` that involves a subset of the variables of the program.

Invariants

An *invariant* is a condition that is guaranteed to be true at a specified set of points p_1, p_2, ..., p_n during the execution of a program. The use of the term "invariant" is designed to emphasize that this condition is true repeatedly. To more precisely illustrate the relationship between an invariant and a program, let `fragment_i` denotes the portion of the program that is executed between the points p_i and p_{i+1}. (Assume `fragment_0` starts at the beginning of the program.) If the invariant holds at each point p_i, we could annotate this trace of the program's execution with `assert` statements between consecutive fragments.

```
fragment_0
assert invariant(...)
fragment_1
assert invariant(...)
...
assert invariant(...)
fragment_n
```

Well-designed invariants help to model the logical process underlying a computation and can be used to verify that the computation has produced a correct answer. Given an appropriate invariant, we can use this invariant in designing the code fragment `fragment_i`. In particular, we need to design each fragment with the following philosophy: **Assume that the invariant is true prior to the fragment. Write a code fragment that moves the computation towards the final answer and ensures that the invariant is again true at the end of the fragment**. Note that if each fragment satisfies this design property, then the entire chain of fragments must satisfy this property.

Example - Loop invariants

To illustrate the use of invariants, we have prepared this program that includes the examples below and more. The first example is the iterative version of the factorial function below.

```
def iterative_factorial(num):
    answer = 1
    index = 0
    assert answer == math.factorial(index)
    while index < num:
        index += 1
        answer *= index
        assert answer == math.factorial(index)
```

```
    return answer
```

Note that we have already annotated this function with the invariant expression `answer == math.factorial(index)`. Observe that this invariant is initially true after execution of the code fragment

```
answer = 1
index = 0
```

During the execution of the body of the `while` loop, the fragment

```
index += 1
answer *= index
```

ensures that the invariant remains true as `index` increases. Finally, note that, when `index == num`, the loop terminates. So, the final instance of the invariant is that `answer == math.factorial(num)`, verifying that the function has correctly computed the desired factorial.

Example - Invariants for recursive functions

Invariants can also be used to reason about the correctness of recursive programs. Consider the following recursive program for computing the factorial function.

```
def recursive_factorial(num):
    if num == 0:
        answer = 1
        assert answer == math.factorial(num)
        return answer
    else:
        rec_part = recursive_factorial(num - 1)
        answer = num * rec_part
        assert answer == math.factorial(num)
        return answer
```

Note that the code fragment for the base case ensures that `num == 0` and `answer == 1`, which ensures that the invariant `answer == math.factorial(num)` is true. For the recursive case, if we assume that the invariant is true prior to execution of the code fragment,

```
rec_part = recursive_factorial(num - 1)
answer = num * rec_part
```

then the invariant `answer == math.factorial(num)` is again true after execution of this fragment.

Example - Class invariants

Invariants are also useful in capturing the relationships between the various data stored in an object. As our last example, recall that the `WildFire` class for our wild fire demo created objects that include both a grid and a queue. During the BFS search in this demo, the program maintained the invariant that every cell in the fire's boundary also had its corresponding grid cell set to full.

Since this condition cannot be easily captured by a single Boolean expression, we have implemented the invariant as a new class method `boundary_invariant` for the `WildFire` class.

```
    def boundary_invariant(self):
        for cell in self.fire_boundary():
            if self.is_empty(cell[0], cell[1]):
                print "Cell " + str(cell) + " in fire boundary is empty."
                return False
        return True
```

Note this method checks the invariant for cells in the fire's boundary and, if it detects a violation of the invariant, prints a message and returns false. In our example code, we have also added a check of the invariant after the body of `update_boundary`.

This example with the `WildFire` class illustrates the importance of explicitly formulating invariants for your classes. In our implementation of the `WildFire` class, we carelessly designed the `enqueue_boundary` method to enqueue a cell on the fire's boundary without setting its corresponding cell in the grid to be full. Instead, we relied on the user of the class to maintain this invariant by making a call to `set_full` after an enqueue. Explicitly formulating the class invariant suggests a better design for the class in which `enqueue_boundary` also sets the corresponding cell in the grid to be full.

Created Mon7 Jul 20149:54 PM BST

Last Modified Sat 26 Jul 2014 5:44 PM BST