EPFL

Identity and Change

Principles of Functional Programming

Martin Odersky

Identity and Change

Assignment poses the new problem of deciding whether two expressions are "the same"

When one excludes assignments and one writes:

```
val x = E; val y = E
```

where `E` is an arbitrary expression, then it is reasonable to assume that `x` and `y` are the same. That is to say that we could have also written:

```
val x = E; val y = x
```

(This property is usually called *referential transparency*)

Identity and Change (2)

But once we allow the assignment, the two formulations are different. For example:

```
val x = BankAccount()
val y = BankAccount()
```

Question: Are `x` and `y` the same?

0 Yes
0 No

Operational Equivalence

To respond to the last question, we must specify what is meant by "the same".

The precise meaning of "being the same" is defined by the property of *operational equivalence*.

In a somewhat informal way, this property is stated as follows.

Suppose we have two definitions `x` and `y`.

`x` and `y` are operationally equivalent if *no possible test* can distinguish between them.

Testing for Operational Equivalence

To test if `x` and `y` are the same, we must

- Execute the definitions followed by an arbitrary sequence `f` of operations that involves `x` and `y`, observing the possible outcomes.

```
val x = BankAccount()
val y = BankAccount()
S
```

Testing for Operational Equivalence

To test if `x` and `y` are the same, we must

- Execute the definitions followed by an arbitrary sequence of operations that involves `x` and `y`, observing the possible outcomes.

```
val x = BankAccount()
val y = BankAccount()
S
```

```
val x = BankAccount()
val y = BankAccount()
S' = [x/y]S
```

- Then, execute the definitions with another sequence `S'` obtained by renaming all occurrences of `y` by `x` in `S`

Testing for Operational Equivalence

To test if `x` and `y` are the same, we must

- Execute the definitions followed by an arbitrary sequence of operations that involves `x` and `y`, observing the possible outcomes.

```
val x = BankAccount()
val y = BankAccount()
S
```

```
val x = BankAccount()
val y = BankAccount()
S' = [x/y]S
```

- Then, execute the definitions with another sequence `S'` obtained by renaming all occurrences of `y` by `x` in `S`
- If the results are different, then the expressions `x` and `y` are certainly different.

Testing for Operational Equivalence

To test if `x` and `y` are the same, we must

- Execute the definitions followed by an arbitrary sequence of operations that involves `x` and `y`, observing the possible outcomes.

```
val x = BankAccount()
val y = BankAccount()
S
```

```
val x = BankAccount()
val y = BankAccount()
S' = [x/y]S
```

- Then, execute the definitions with another sequence `S'` obtained by renaming all occurrences of `y` by `x` in `S`
- If the results are different, then the expressions `x` and `y` are certainly different.
- On the other hand, if all possible pairs of sequences `(S, S')` produce the same result, then `x` and `y` are the same.

Counterexample for Operational Equivalence

Based on this definition, let's see if the expressions

```
val x = BankAccount()
val y = BankAccount()
```

define values `x` and `y` that are the same.

Let's follow the definitions by a test sequence:

```
val x = BankAccount()
val y = BankAccount()
x.deposit(30)                    // : Int = 30
y.withdraw(20)                   // java.lang.Error: insufficient funds
```

Counterexample for Operational Equivalence (2)

Now rename all occurrences of `y` with `x` in this sequence. We obtain:

```
val x = BankAccount()
val y = BankAccount()
x.deposit(30)                    // : Int = 30
x.withdraw(20)                   // : Int = 10
```

The final results are different. We conclude that `x` and `y` are not the same.

Establishing Operational Equivalence

On the other hand, if we define

```
val x = BankAccount()
val y = x
```

then no sequence of operations can distinguish between `x` and `y`, so `x` and `y` are the same in this case.

Assignment and Substitution Model

The preceding examples show that our model of computation by substitution cannot be used.

Indeed, according to this model, one can always replace the name of a value by the expression that defines it. For example, in

```
val x = BankAccount()
val y = x
```

the `x` in the definition of `y` could be replaced by `BankAccount()`

Assignment and The Substitution Model

The preceding examples show that our model of computation by substitution cannot be used.

Indeed, according to this model, one can always replace the name of a value by the expression that defines it. For example, in

```
val x = BankAccount()
val y = x
```

```
val x = BankAccount()
val y = BankAccount()
```

the `x` in the definition of `y` could be replaced by `BankAccount()`

But we have seen that this change leads to a different program!

The substitution model ceases to be valid when we add the assignment.

It is possible to adapt the substitution model by introducing a *store*, but this becomes considerably more complicated.