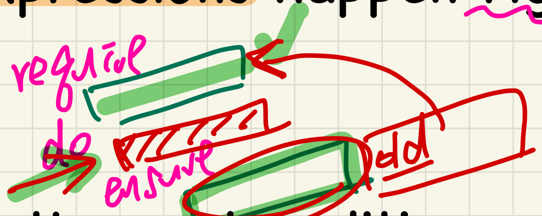
EECS3311 Software Design (Fall 2020)

Q&A - Lecture Series W2

Tuesday, September 22

Q. Does the caching of the old expressions happen right after the syntax checks?

X



Q. Somehow, the compiler checks the postconditions and before evaluating them, it finds all the uses of the old keyword and caches them before the implementation. Is this correct?

Q. Is this why `old get (j.item)` doesn't work? Since the postcondition has not been evaluated yet, `j` doesn't receive an integer to work with.

Use of old in across Expression in Postcondition

```
class LINEAR_CONTAINER
create make
feature -- Attributes
  a: ARRAY[STRING]
feature -- Queries
  count: INTEGER do Result := a.count end
  get (i: INTEGER): STRING do Result := a[i] end
feature -- Commands
  make do create a.make_empty end
  update (i: INTEGER; v: STRING)
  do ...
  ensure -- Others Unchanged
    across
      1 |..| count as j
    all
      j.item /= i implies old get(j.item) ~ get(j.item)
    end
  end
end
```

old-get-j-item := get(j.item)

X

Hint: What value will be cached at runtime

before executing the implementation of update?

Use of old in across Expression in Postcondition

```
class LINEAR_CONTAINER
create make
feature -- Attributes
  a: ARRAY[STRING]
feature -- Queries
  count: INTEGER do Result := a.count end
  get (i: INTEGER): STRING do Result := a[i] end
feature -- Commands
  make do create a.make_empty end
  update (i: INTEGER; v: STRING)
  do ...
  ensure -- Others Unchanged
      across
        1 |..| count as j
      all
        j.item /= i implies old get(j.item) ~ get(j.item)
      end
  end
end
```

Q. Also, can't we just cache the
a: ARRAY
and check against that?

(old Current.deep_twin).get(j.item)

vs.

old Current.deep_twin.get(j.item)

→ to be cached in the pre-state

→ still doesn't exist
↳ compilation prob.

Revisit: Bank Accounts in Java V5

```
public class AccountV5 {
  public void withdraw(int amount) throws
    WithdrawAmountNegativeException, WithdrawAmountTooLargeException {
   int oldBalance = this.balance;
   if(amount < 0) { /* negated precondition */
     throw new WithdrawAmountNegativeException(); }
   else if (balance < amount) { /* negated precondition */
     throw new WithdrawAmountTooLargeException(); }
   else { this.balance = this.balance - amount; }
   assert this.getBalance() > 0 :"Invariant: positive balance";
    assert this.getBalance() == oldBalance - amount :
     "Postcondition: balance deducted"; }
```

pre-state

manual

post-state

pre-state

How does the corresponding Eiffel design look like (with **automatic** caching of **pre-state** values)?

withdraw (a)

ensure

$\boxed{\text{Current.balance}} = \text{old balance} - a$

Q. When would reference copy be appropriate?

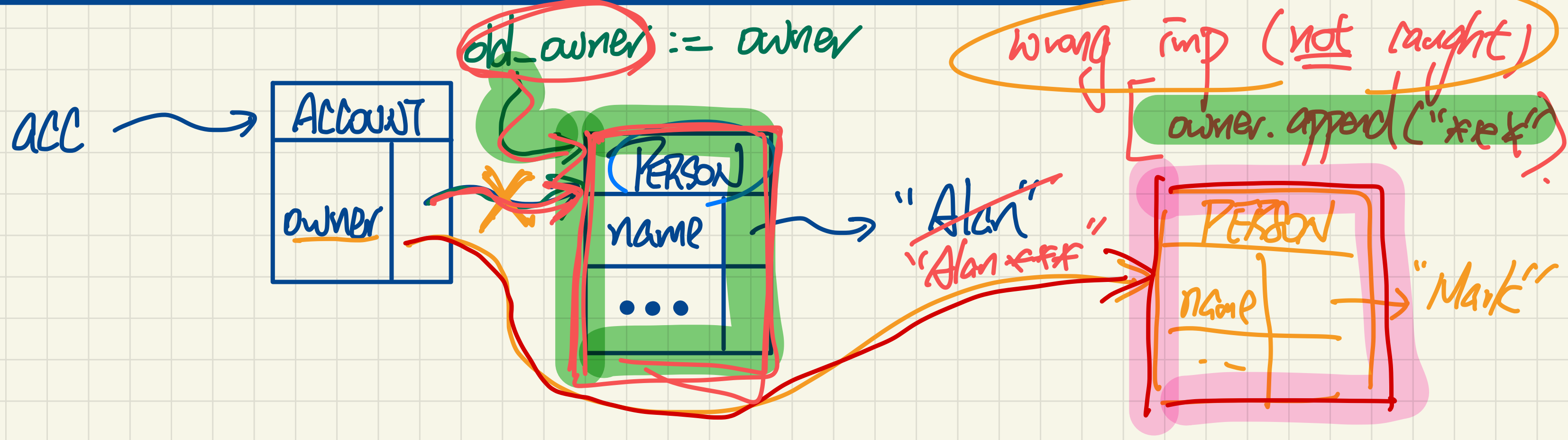
Is it just for primitive objects?

```
class ACCOUNT
  owner: PERSON
  withdraw(...)
    ensure
      same_person: owner = old owner
      ~~equal_person: owner ~ old owner.deep_twin~~
end
```

postcondition violation (good)

wrong imp: (caught)

```
owner := create {PERSON}.make ("Mark")
```



Q. When would reference copy be appropriate?

Is it just for primitive objects?

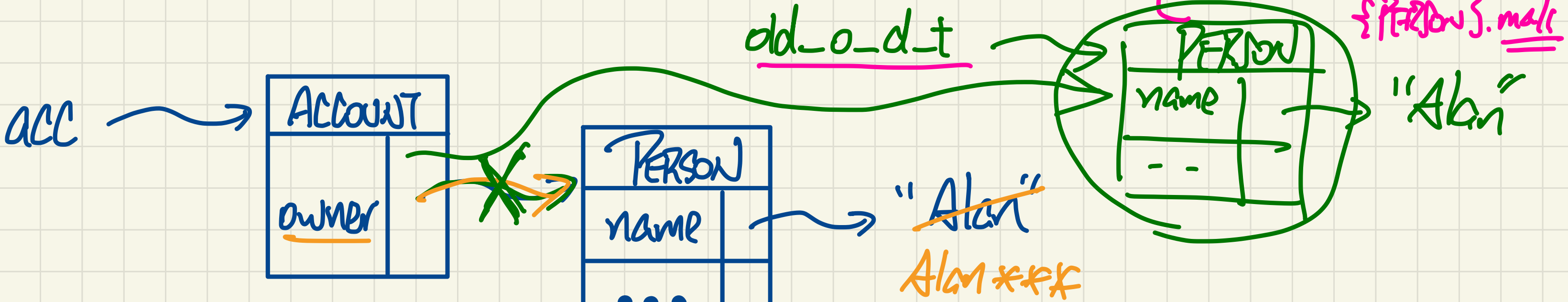
```
class ACCOUNT
  owner: PERSON
  withdraw(...)
    ensure
      ~~same_person: owner = old owner~~
      equal_person: owner ~ old owner.deep_twin
end
```

Wrong Imp (caught) → postcond. violation

owner.name.append("***")

Wrong Imp

owner := create {PERSON}.make



Rather than doing `Current.account_of (acc.owner)`, could we use a 2nd across statement for iterating over the post-state version of `accounts`?

```
others_unchanged :
  across old accounts.deep_twin is acc
  all
    acc.owner /~ n implies acc ~ Current.account_of(acc.owner)
  end
```

Something like:

pre-state

unnecessary

```
across old accounts.deep_twin is pre_acc all
across accounts is post_acc all
    pre_acc.owner /~ n and pre_acc.owner ~ post_acc.owner
    implies
    pre_acc ~ post_acc
end
end
```

ϕ i j

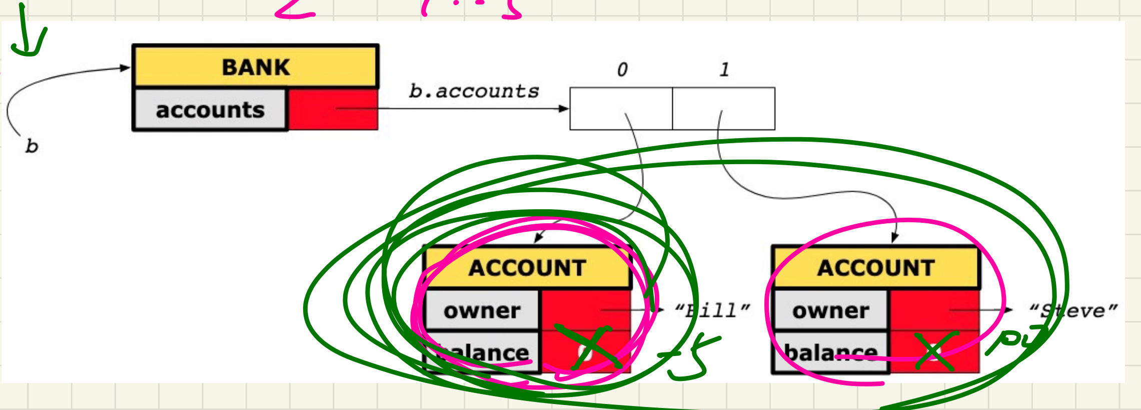
1 1..5

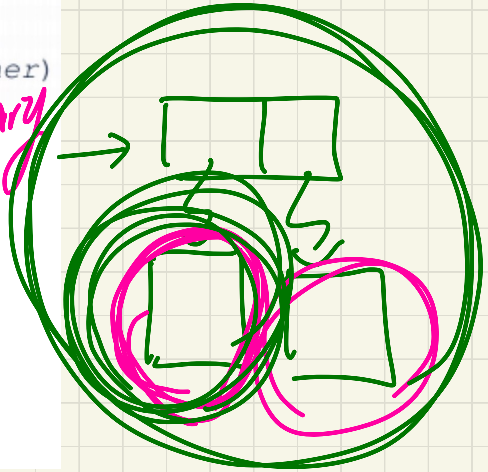
2 1..5

```
across |1..|10| is i loop
across |1..|5| is j loop
  [ i , j ]
end
end
```

10.

5.





Writing Postcondition: Exercise

```
is_positive (i: INTEGER): BOOLEAN
  do
    Result := i > 0
  ensure
    i > 0
```

T F

require i > 0

is_positive(0)
↳ Result := false

② if i > 0 then
Result
else
not Result
end

too strong

is_positive(2)
3
4

① i > 0 implies Result = True
and ¬(i>0)
i <= 0 implies Result = False

Result

not Result

③ Result = i > 0

Writing Postcondition: Exercise

```
is_positive (x: INTEGER): BOOLEAN
  ensure
    case_1: x > 0 and Result = True
    case_2: x <= 0 and Result = False
```

implies

implies

$P \wedge \neg P$ F

Positive

is_positive(3) → TRUE

↳ case_1 : 3 > 0 and T = T   T

and

case_2 : 3 <= 0 and T = F   F

postcond violation

is_positive(-3) R=F

↳ case_1 : -3 > 0 ⇒ F = T

F   T

Non-Positive

↳ case_2 : -3 <= 0 T

⇒ F = F   T

is_positive (-3)   False

↳ case_1 : -3 > 0 and F = T

T ⇒ T   T   F   F → postcond-violation

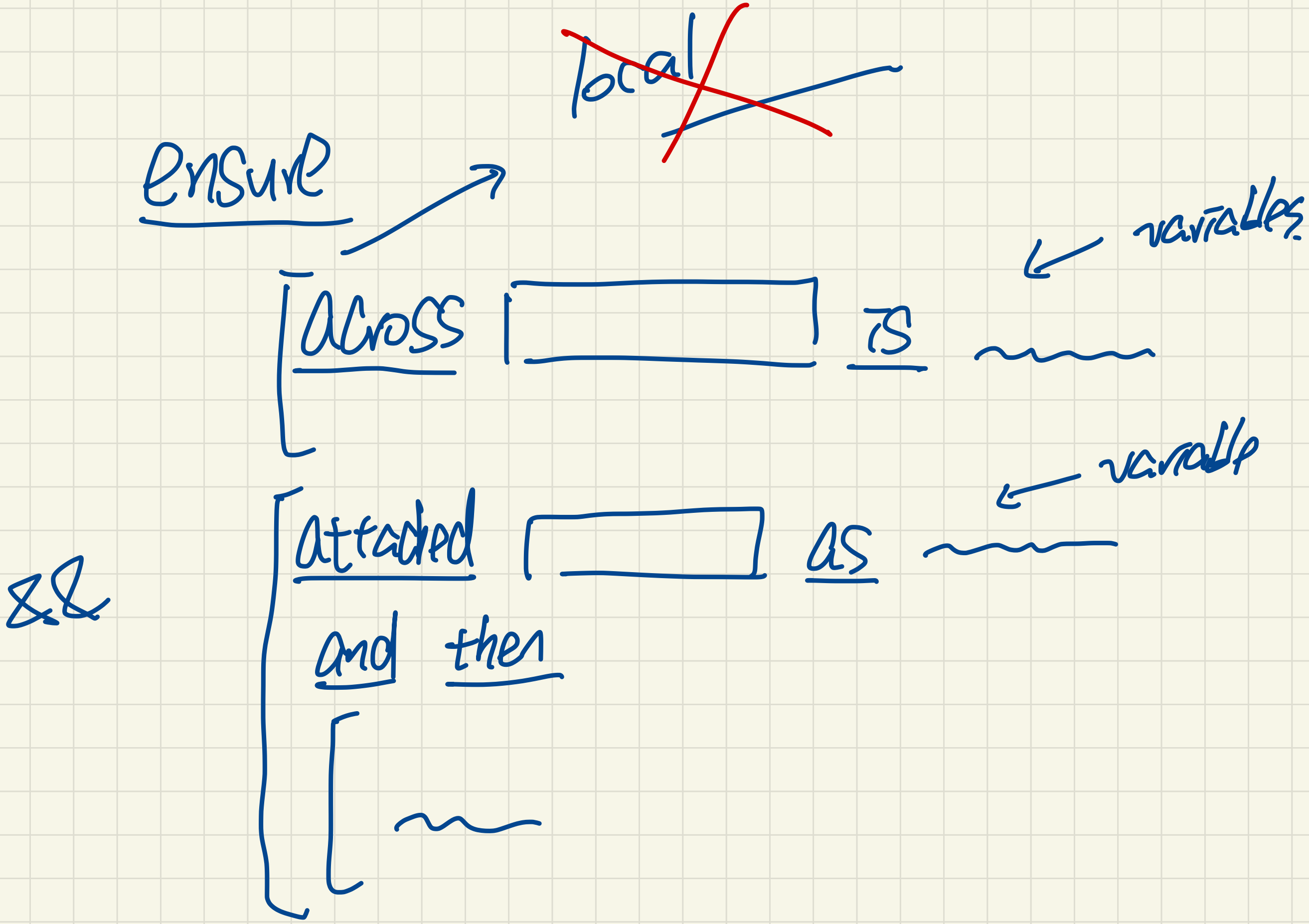
Writing Postcondition: Exercise

```
is_positive (x: INTEGER): BOOLEAN
  ensure
    if x > 0 then
      Result = True
    end
```

compilation error

else
Result = F

ensure
if then
else
else
X


~~local~~
ensure
across
is
variables
attached
as
variable
&&
and then