Lecture 8: Contracts and Inheritance
(based on work with Piotr Nienaltowski)
Preconditions

- In sequential context: precondition is correctness condition
- In concurrent context: feature call and feature application do not usually coincide
  - A supplier cannot assume that a property satisfied at the call time still holds at the execution time.
store (b: separate BUFFER [INTEGER]; i: INTEGER)
    -- Store i in buffer.
    require
        not b.is_full
        i > 0
    do
        b.put (i)
    end

my_buffer: separate BUFFER [INTEGER]
ns_buffer: BUFFER [INTEGER]
...
store (my_buffer, 24)
store (ns_buffer, 24)
my_buffer := ns_buffer
store (my_buffer, 79)
Preconditions

• A precondition expresses the necessary requirements for a correct feature application.

• Precondition viewed as synchronization mechanism:
  • A called feature cannot be executed unless the preconditions hold
  • A violated precondition delays the feature’s execution

• The guarantee given to the supplier is exactly the same as with the traditional semantics.
Postconditions

• A postcondition describes the result of a feature’s application.
• Postconditions are evaluated asynchronously; wait by necessity does not apply.
• Postcondition clauses that do not involve calls on objects handled by the same processors are evaluated independently.
Postconditions

**spawn_two** (l1, l2: separate LOCATION)
  do
    l1.do_job
    l2.do_job
  ensure
    post_1: l1.is_ready
    post_2: l2.is_ready
  end

tokyo, zurich: separate LOCATION

r (l: separate LOCATION)
  do
    spawn_two (l, tokyo)
    do_local_stuff
    get_result (l)
    do_local_stuff
    get_result (tokyo)
  end

... 
r (zurich)
Invariants

- Standard Eiffel semantics applies to class invariants in SCOOP because all the calls appearing in invariants must be non-separate.
Loop Invariants, Loop Variants and Check

- Asynchronous semantics is applied to loop variants, loop invariants, and check instruction.
Loop Invariants and Loop Variants

```
remove_n (list: separate LIST [G]; n: INTEGER)
  -- Remove n elements.
require
  list.count >= n
local
  initial, removed: INTEGER
do
  from
    initial := list.count
    removed := 0
until removed = n invariant
  list.count + removed = initial
variant
  list.count - initial + n  -- Same as n - removed
loop
  list.remove
  removed := removed + 1
end
ensure
  list.count = old list.count + 1
end
```
Reasoning about Objects: Sequential

\(\{\text{INV} \land \text{Pre}_r\} \text{ body}_r \{\text{INV} \land \text{Post}_r\}\)

\[\frac{\{\text{Pre}_r\} \times \text{r} (a) \{\text{Post}_r\}}{}\]

Only n proofs if n exported routines!
Reasoning about Objects: Concurrent

\{INV \land Pre_r\} \text{ body}_r \{INV \land Post_r\}

\{Pre_r^{\text{cont}}\} \times r (a) \{Post_r^{\text{cont}}\}

- Hoare-style sequential reasoning
- Controlled expressions (known statically as part of the type system) are both:
  - Attached (statically known to be non-void)
  - Handled by processor locked in current context
Inheritance

• Can we use inheritance as in the sequential world?
• Is multiple inheritance allowed?
• Does SCOOP suffer from inheritance anomalies?
Example: Dining Philosophers

class PHILOSOPHER inherit GENERAL_PHILOSOPHER
  PROCESS
  rename
    setup as getup
  undefine
    getup
end

feature
  step
    -- Perform a philosopher's tasks.
    do
      think ; eat (left, right)
    end

  eat (l, r: separate FORK)
    -- Eat, having grabbed l and r.
    do ... end
end
Dining Philosophers

defered class PROCESS feature
  over: BOOLEAN
    -- Should execution terminate now?
    deferred end

setup
  -- Prepare to execute process operations.
  deferred end

step
  -- Execute basic process operations.
  deferred end

wrapup
  -- Execute termination operations (default: nothing).
  do end

live
  -- Perform process lifecycle.
  do
    from setup until over loop
      step
    end
    wrapup
  end end
Dining Philosophers

class GENERAL_PHILOSOPHER create
    make
feature -- Initialization
    make (l, r: separate FORK)
        -- Define l as left and r
        -- as right forks.
    do
        left := l
        right := r
    end
end

class FORK
end

feature {NONE} -- Implementation
    left: separate FORK
    right: separate FORK
    getup
        -- Take initialization actions.
        do end
    think
        -- Philosopher’s act.
        do end
end
Inheritance

• Full support for inheritance (including multiple inheritance)
• Most inheritance anomalies eliminated thanks to the proper use of OO mechanisms
• What’s about Active objects?
Inheritance and Contracts

- Preconditions may be kept or weakened.
  - Less waiting
- Postconditions may be kept or strengthened.
  - More guarantees to the client
- Invariants may be kept or strengthened
  - More consistency conditions
Inheritance: Result redeclaration

- Covariant result redeclaration causes no problems. The client waits less.
Inheritance: formal argument redecleration

- Formal argument types may be redefined contravariantly w.r.t processor tag.
- The client may only use non-separate actual arguments in polymorphic calls, so no additional blocking will happen.
• Formal argument types may be redefined contravariantly w.r.t detachable tags. The client waits less.
Effect on contracts

- Inherited precondition and postcondition clauses that involve calls on the redefined arguments may become invalid.

```plaintext
class A feature
  s (x: separate X)
  require
    x.some_precondition
do
  ...
ensure
  x.some_postcondition
end
end
class B inherit A redefine s end
feature
  s (x: detachable separate X)
do
  ...
ensure
  x.some_postcondition
end
end
```

The call on `x` in the inherited pre- and postcondition is invalid.
Effect on contracts

• The redeclaring feature cannot rely on the precondition clauses involving the formal argument.
• Inherited precondition clauses involving calls on detachable formal arguments hold vacuously (trivially true).
• Postconditions involving detachable formal arguments cannot be allowed.
Feature Redeclaration

• The redefinition rules for the individual components of a type are now clarified
  • Class types may be redefined covariantly both in result types and argument types but the redefinition of a formal argument forces it to become detachable. For example, assuming that \( Y \) conforms to \( X \), an argument \( x: X \) may be redefined into \( x: \text{detachable} \ Y \) but not \( x: Y \). An attribute may be redefined from \( \text{my}_x: X \) into \( \text{my}_x: Y \).
  • Detachable tags may be redefined from \( ? \) to \( ! \) in result types. They may be changed from \( ! \) to \( ? \) in argument types, provided that no call on the redefined argument occurs in the original postcondition.
  • Processor tags may be redefined from \( T \) to something more specific in result types, and from more specific to \( T \) in argument types.
Problems with Precursor Calls

- **Use of Precursor is not always possible.**

```plaintext
class A
feature
  r (x: separate X; y: separate Y)
    require
      some_precondition
    do
      x.f
      y.f
    end
end
class B
inherit A
redefine r end
feature
  r (x: separate X; y: detachable separate Y)
    require else
      new_precondition
    do
      if new_precondition then
        -- do something here
      else
        Precursor (x, y)
      end
    end
end
```

Here some_precondition holds

invalid
Solution for Precursor Calls

Need object test (downcast):

```plaintext
class B inherit
  A redefine r end
feature
  r (x: separate X; y: detachable separate Y)
  require else
    new_precondition
  do
    if new_precondition then
      ...
    elseif attached y as ly: separate Y then
      Precursor (x, ly)
  end
end
end
```