ETH Zürich
Chair of Software Engineering

Concepts of Concurrent Computation

Bertrand Meyer
Sebastian Nanz

Lecture 8: SCOOP advanced concepts

Today's lecture

In this lecture you will learn about:

- Lock passing, a mechanism implemented in SCOOP for deadlock avoidance
- The changed semantics of contracts in SCOOP, especially that of postconditions
- Inheritance in SCOOP
- Definition and use of agents (function objects) in SCOOP
- The semantics of once functions in SCOOP

EiffelSoftware SCOOP capabilities

- Processor tags: not supported
- Asynchronous postcondition evaluation: not supported
- (Deep) Import operation for expanded types: not supported
- Lock passing: supported
- Separate callbacks: not supported
- Valid feature redeclaration with respect to separateness: supported
- Object tests that incorporate processor locality: not supported
- Agents: not supported
- Once routines: supported

Lock passing

The need for lock passing

```
r (x: separate X; y: separate Y)
    local
        z: separate ANY
    do
        x.f
        x.g (y)
        y.f
        z := x.some_query
    end
```

y is locked by **Current**

Waits for y to become available

Deadlock: wait for some_query to finish

Lock passing

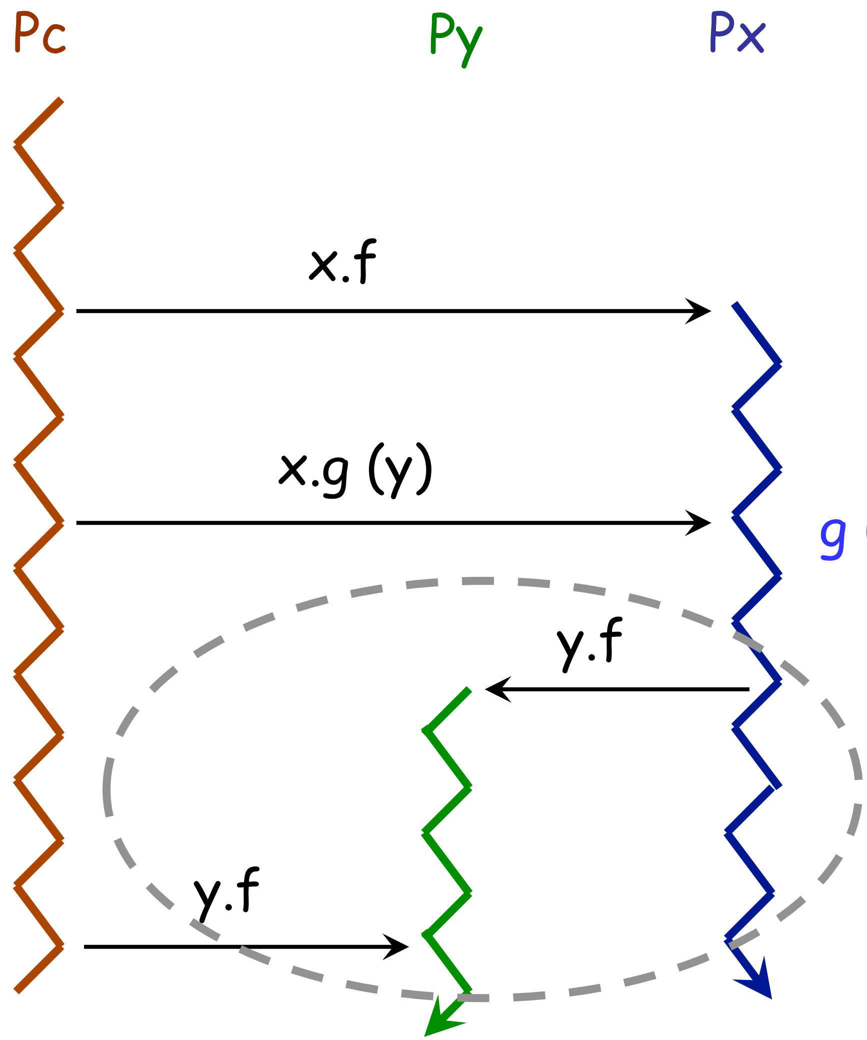
...

x.f

x.g (y)

...

y.f



g (y: separate Y)
    do
        y.f
        ...
        ...
        ...
    end

Lock passing

- If a call x.f (a_1, … , a_n) occurs in a routine r where one or more a_i are controlled, the client's handler (the processor executing r) passes all currently held locks to the handler of x, and waits until f terminates
- When f terminates, the client resumes its computation.

```
r (x: separate X; y: separate Y)
    local
        z: separate ANY
    do
        x.f
        x.g (y)
        y.f
        z := x.some_query
    end
```

Pass locks to g and wait for g to finish

Synchronous

Synchronous

Lock passing combinations

Formal → / ↓ Actual	Attached	Detachable
Reference, controlled	**Lock passing**	no
Reference, uncontrolled	no	no
Expanded	no	no

Lock passing: example

```
class C feature
    x1: X
    z1: separate Z
    c1: separate C
    i: INTEGER

    r (x: separate X; y: separate Y)
        do
            x1.f (5)
            x1.g (x)
            i := x1.h (Current)
            x.f (10)
            x.g (z1)
            x.g (y)
            x.m (y)
            i := x.h (c1)
            i := x.h (Current)
        end
    p (...) do ... end
end
```

```
class X feature
    f (i: INTEGER) do ... end
    g (a: separate ANY) do ... end
    h (c: separate C): INTEGER do c.p (... ) end
    m (a: detachable separate ANY) do ... end
end
```

x1.f (5): Non-separate, no wait by necessity, no lock passing

x1.g (x): Non-separate, no wait by necessity, lock passing (vacuous)

i := x1.h (Current): Non-separate, wait by necessity, lock passing (vacuous)

x.f (10): Separate, no wait by necessity, no lock passing

x.g (z1): Separate, no wait by necessity, no lock passing

x.g (y): Separate, no wait by necessity, lock passing

x.m (y): Separate, no wait by necessity, no lock passing

i := x.h (c1): Separate, wait by necessity, no lock passing

i := x.h (Current): Separate, wait by necessity, lock passing

Contracts

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.

Preconditions

```
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 holds
  - 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.
- After returning from the call the client can only assume the controlled postcondition clauses.

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)
```

Inheritance

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

```
deferred 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
```

```
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

Inheritance and Contracts

- Preconditions may be kept or weakened.
  - Less waiting
- Postconditions may me kept or strengthened.
  - More guarantees to the client
- Invariants may be kept or strengthened
  - More consistency conditions
- See Piotr Nienaltowski, Bertrand Meyer, Jonathan S. Ostroff: *Contracts for concurrency*. Formal Aspects of Computing, 21(4): 305-318 (2009); see se.ethz.ch/~meyer/publications/concurrency/contracts_for_concurrency.pdf

Inheritance: Result type redeclaration (functions)

```
class C feature
    r (x: X)
        do … end

    s (y: separate Y)
        do … end
end
```

```
--Would lead to a traitor:
c: C  a: A
create {B} a

c.r (a.x)
```

```
class A feature
    x: X
    y: separate Y
end
```

```
class B
inherit A redefine x, y end
feature
    x: separate X
    y: Y
end
```

```
-- This one is OK:
c: C    a: A
create {B} a
c.s (a.y)
```

- Result types may be redefined covariantly for functions. For attributes the result type may not be redefined.

Inheritance: formal argument redeclaration

```
class A feature
    r (x: separate X)
        do … end
s (x: X)
        do … end
end
```

```
-- x could be a traitor:
a: A  x: separate X
create {B} a
a.r (x)
```

```
class B inherit
    A redefine r, s end
feature
    r (x: X)
        do … end
    s (x: separate X)
        do … end
end
```

```
-- OK
```

- Formal argument types may be redefined contravariantly w.r.t. processor tags.

Inheritance: formal argument redeclaration

```
class A feature
    r (x: detachable separate X)
        do … end

    s (x: separate X)
        do … end
end
```

```
class B inherit
    A redefine r, s end
feature
    r (x: separate X)
        do … end        -- Additional locking for client: not acceptable

    s (x: detachable separate X)
        do … end        -- Less locking for client: acceptable
end
```

- Formal argument types may be redefined contravariantly w.r.t detachable tags. The client waits less.

Agents

What is an agent?

- An agent represents an operation ready to be called.

```
x: X
op1: ROUTINE [X, TUPLE]


op1 := agent x.f
op1.call ([])
```

- Agents can be created by one object, passed to another one, and called by the latter

What is an agent?

- Arguments can be closed (fixed) or open.

```
op1 := agent io.put_string ("Hello World!")
op1.call ([])
```

Empty tuple as argument

```
op1 := agent io.put_string (?)
op1.call (["Hello World!"])
```

One-argument tuple

- They are based on generic classes:

```
ROUTINE [BASE_TYPE, OPEN_ARGS -> TUPLE]
PROCEDURE [BASE_TYPE, OPEN_ARGS -> TUPLE]
FUNCTION [BASE_TYPE, OPEN_ARGS -> TUPLE, RESULT_TYPE]
```

Use of agents

Object-oriented wrappers for operations

- Strongly-typed function pointers (C++)
- Similar to .NET delegates

Used in event-driven programming

- Subscribe an action to an event type
- The action is executed when event occurs

Loose coupling of software components

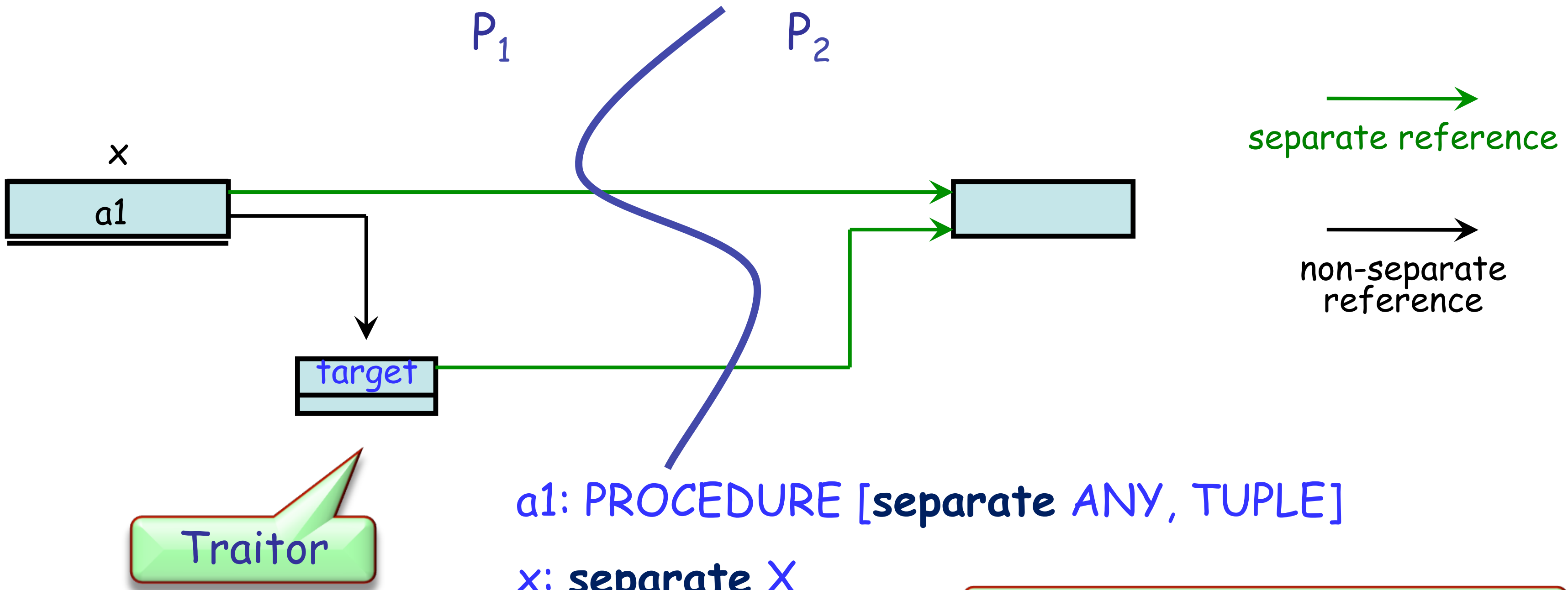
Replace several patterns

- Observer
- Visitor
- Model - View – Controller

. . .

Problematic agents

- Which processor should handle an agent? Is it the target processor or the client processor?
- Let's assume it is the client processor.

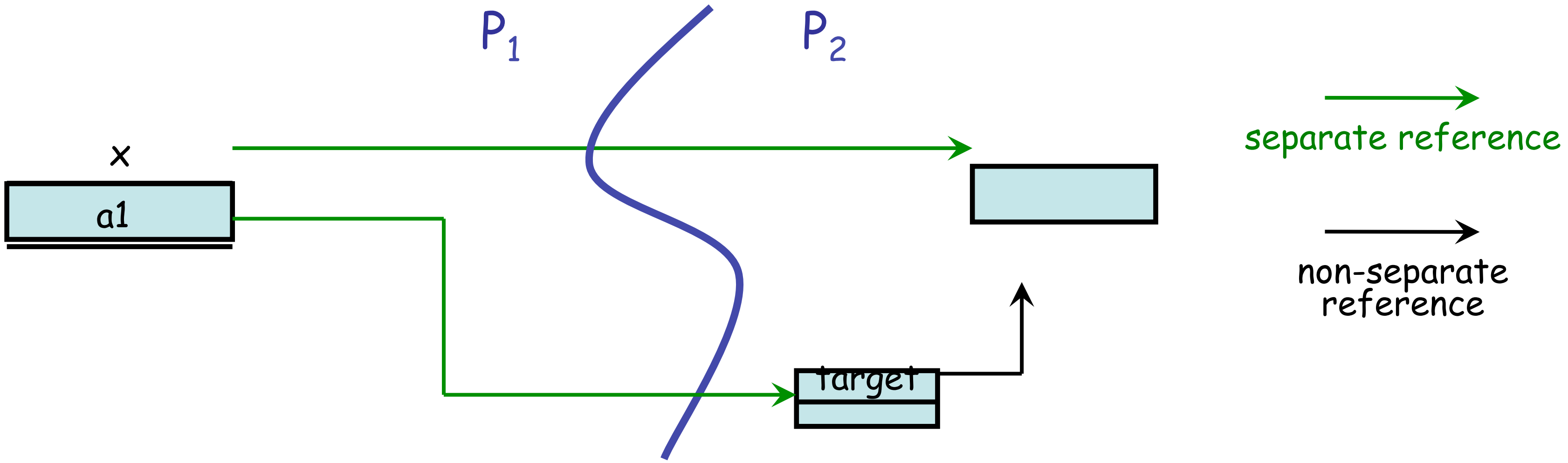



```
a1: PROCEDURE [separate ANY, TUPLE]
x: separate X
. . .
a1 := agent x.f
a1.call ([])
```

Like x.f without locking x

Let's make the agent separate!

- The agent needs to be on the target processor.



a1: **separate** PROCEDURE [X, TUPLE]

x: **separate** X

. . .

a1 := **agent** x.f

a1.call ([])

This agent will be handled by x's processor

Invalid

Let's make the agent separate!

- No special type rules for separate agents
- Semantic rule: an agent is created on its target's processor
- Agents pass processors' boundaries just as other objects do

```
a1: separate PROCEDURE [X, TUPLE]
x: separate X
a1 := agent x.f

call (a1)
call (an_agent: separate PROCEDURE [ANY, TUPLE])
    do
        an_agent.call ([])    -- Valid separate call
    end
```

First benefit: convenience

- Without agents, enclosing routines are necessary for every separate call.

```
x1: separate X
r (x1)
s (x1)
```

```
r (x: separate X)
    do
        x.f
    end
```

```
s (x: separate X)
    do
        x.g (5, "Hello")
    end
```

- With agents, we can write a universal enclosing routine.

```
call (agent x1.f); call (agent x1.g (5, "Hello"))

call (an_agent: separate PROCEDURE [ANY, TUPLE])
        -- Universal enclosing routine.
    do
        an_agent.call ([])
    end
```

Second benefit: full asynchrony

- Without agents, full asynchrony cannot be achieved

```
x1, y1: separate X
r (x1)              -- Blocking
do_local_stuff
```

```
r (x: separate X)
    do
        x.f         -- Asynchronous
    end
```

- With agents it works

```
asynch (agent x1.f)    -- Non-blocking
do_local_stuff
```

```
asynch (a: detachable separate PROCEDURE [ANY, TUPLE])
        -- Call a asynchronously.
    do
        . . .
    end
```

Full asynchrony

The feature asynch can be implemented as follows:

```
asynch (a : detachable separate PROCEDURE [ANY, TUPLE])
        -- Call a asynchronously.
        -- Note that a is not locked.
    local
        executor: separate EXECUTOR
    do
        create executor.make (a)
        launch    (executor)
    end
```

An asynchronous call on a non-separate targets (including **Current**) will be executed when the current processor becomes idle.

Third benefit: waiting faster

```
x1, y1: separate X

if or_else (x1, y1) then
    . . .
end
```

```
or_else (x, y: separate X): BOOLEAN
    do
        Result := x.b or else y.b
    end
```

- What if x1 or y1 is busy?
- What if x1.b is false but y1.b is true?
- What if evaluation of x1.b takes ages whereas y1.b evaluates very fast?

Waiting faster

```
if parallel_or (agent x1.b, agent y1.b) then
    ...
end

parallel_or (a1, a2: detachable separate FUNCTION [ANY, TUPLE, BOOLEAN]): BOOLEAN
        -- Result of a1 or else a2 computed in parallel.
    local
        ans_col: separate ANSWER_COLLECTOR [BOOLEAN]
    do
        create ans_col.make (a1, a2)
        Result := answer (ans_col)
    end
answer (ac: separate ANSWER_COLLECTOR [BOOLEAN]): BOOLEAN
        -- Result returned by ac.
    require
        answer_ready: ac.is_ready
    do
        Result ?= ac.answer
    end
```

Agents wrap-up

- Agents and concurrency
  - Tricky at first; easy in the end
  - Agents built on separate calls are separate
  - Agents treated just like any other object
- Advantages brought by agents
  - Convenience: "universal" enclosing routine for single calls
  - Full asynchrony: non-blocking calls
  - Truly parallel wait

Once functions

Once Functions

- Similar to constants
    - Always return the same value
- Lazy evaluation
    - Body executed on first access
- Once per thread or once per object semantic
- Examples of use
    - Heavy computations
        - Stock market statistics
    - Common contact point for objects of one type
        - Feature io in class ANY

Once functions in a concurrent context

- Is once-per-system semantics always correct?

```
barrier: separate BARRIER
    once
        create Result.make (3)
    end
```

```
local_printer: PRINTER
    once
        printer_pool.item (Current.location)
    end
```

- Separate functions are once-per-system.
- Non-separate functions are once-per-processor.